# To Hash or Not to Hash Again? (In)differentiability Results for $H^2$ and HMAC

Yevgeniy Dodis[*] Thomas Ristenpart[†] John Steinberger[‡] Stefano Tessaro[§]

July 9, 2012

**Abstract**

We show that the second iterate $H^2(M) = H(H(M))$ of a random oracle $H$ cannot achieve strong security in the sense of indifferentiability from a random oracle. We do so by proving that indifferentiability for $H^2$ holds only with poor concrete security by providing a lower bound (via an attack) and a matching upper bound (via a proof requiring new techniques) on the complexity of any successful simulator. We then investigate HMAC when it is used as a general-purpose hash function with arbitrary keys (and not as a MAC or PRF with uniform, secret keys). We uncover that HMAC's handling of keys gives rise to two types of weak key pairs. The first allows trivial attacks against its indifferentiability; the second gives rise to structural issues similar to that which ruled out strong indifferentiability bounds in the case of $H^2$. However, such weak key pairs do not arise, as far as we know, in any deployed applications of HMAC. For example, using keys of any fixed length shorter than $d-1$, where $d$ is the block length in bits of the underlying hash function, completely avoids weak key pairs. We therefore conclude with a positive result: a proof that HMAC is indifferentiable from a RO (with standard, good bounds) when applications use keys of a fixed length less than $d-1$.

**Keywords:** Indifferentiability, Hash functions, HMAC.

[*] Department of Computer Science, New York University, `dodis@cs.nyu.edu`
[†] Department of Computer Sciences, University of Wisconsin–Madison, `rist@cs.wisc.edu`
[‡] Institute of Theoretical Computer Science, Tsinghua University, `jpsteinb@gmail.com`
[§] CSAIL, Massachusetts Institute of Technology, `tessaro@csail.mit.edu`

## Contents


**1 Introduction** **3**
- 1.1 The Second Iterate Paradox . . . 3
- 1.2 HMAC with Arbitrary Keys . . . 5
- 1.3 Discussion . . . 6
- 1.4 Prior Work . . . 6

**2 Preliminaries** **6**

**3 Second Iterates and their Security** **8**
- 3.1 Hash chains using Second Iterates . . . 8
- 3.2 An Example (Vulnerable) Application: Mutual Proofs of Work . . . 11
- 3.3 An Indifferentiability Distinguisher for any Second Iterate . . . 12
- 3.4 Indifferentiability Upper Bound for a Second Iterate . . . 14

**4 HMAC as a General-purpose Keyed Hash Function** **18**
- 4.1 Weak Key Pairs in HMAC . . . 19
- 4.2 Colliding Key Pairs and the Indifferentiability of HMAC . . . 20
- 4.3 Ambiguous Key Pairs and the Indifferentiability of HMAC . . . 21
- 4.4 Indifferentiability Upper Bound for HMAC with Restricted Keys . . . 25

**A Internal Collision Probabilities** **34**

**B Proof of Theorem 3.2** **35**

**C Proof of Theorem 4.2** **40**

**D Proof of Theorem 3.3** **43**

# 1 Introduction

Cryptographic hash functions such as those in the MD and SHA families are constructed by extending the domain of a fixed-input-length compression function via the Merkle-Damgård (MD) transform. This applies some padding to a message and then iterates the compression function over the resulting string to compute a digest value. Unfortunately, hash functions built this way are vulnerable to extension attacks that abuse the iterative structure underlying MD [24, 40]: given the hash of a message $H(M)$ an attacker can compute $H(M \parallel X)$ for some arbitrary $X$, even without knowing $M$.

In response, suggestions for shoring up the security of MD-based hash functions were made. The simplest is due to Ferguson and Schneier [22], who advocate a hash-of-hash construction: $H^2(M) = H(H(M))$, the second iterate of $H$. An earlier example is HMAC [5], which similarly applies a hash function $H$ twice, and can be interpreted as giving a hash function with an additional key input. Both constructions enjoy many desirable features: they use $H$ as a black box, do not add large overheads, and appear to prevent the types of extension attacks that plague MD-based hash functions.

Still, the question remains whether they resist other attacks. More generally, we would like that $H^2$ and HMAC behave like random oracles (ROs). In this paper, we provide the first analysis of these functions as being indifferentiable from ROs in the sense of [14, 33], which (if true) would provably rule out most structure-abusing attacks. Our main results surface a seemingly paradoxical fact, that the hash-of-hash $H^2$ cannot be indifferentiable from a RO with good bounds, *even* if $H$ is itself modeled as a keyed RO. We then explore the fall out, which also affects HMAC.

Indifferentiability. Coron et al. [14] suggest that hash functions be designed so that they "behave like" a RO. To define this, they use the indifferentiability framework of Maurer et al. [33]. Roughly, this captures that no adversary can distinguish between a pair of oracles consisting of the construction (e.g., $H^2$) and its underlying ideal primitive (an ideal hash $H$) and the pair of oracles consisting of a RO and a simulator (which is given access to the RO). A formal definition is given in Section 2. Indifferentiability is an attractive goal because of the MRH composition theorem [33]: if a scheme is secure when using a RO it is also secure when the RO is replaced by a hash construction that is indifferentiable from a RO. The MRH theorem is widely applicable (but not ubiquitously, c.f., [37]), and so showing indifferentiability provides broad security guarantees.

While there exists a large body of work showing various hash constructions to be indifferentiable from a RO (c.f., [1, 7, 12–14, 18, 19, 26]), none have yet analyzed either $H^2$ or HMAC. Closest is the confusingly named HMAC construction from [14], which hashes a message by computing $H^2(0^d \parallel M)$ where $H$ is MD using a compression function with block size $d$ bits. This is not the same as HMAC proper nor $H^2$, but seems close enough to both that one would expect that the proofs of security given in [14] apply to all three.

## 1.1 The Second Iterate Paradox

Towards refuting the above intuition, consider that $H^2(H(M)) = H(H^2(M))$. This implies that an output of the construction $H^2(M)$ can be used as an intermediate value to compute the hash of the message $H(M)$. This property does not exist in typical indifferentiable hash constructions, which purposefully ensure that construction outputs are unlikely to coincide with intermediate values. However, and unlike where extension attacks apply (they, too, take advantage of outputs being intermediate values), there are no obvious ways to distinguish $H^2$ from a RO.

Our first technical contribution, then, is detailing how this structural property might give rise to vulnerabilities. Consider computing a hash chain of length $\ell$ using $H^2$ as the hash function. That is, compute $Y = H^{2\ell}(M)$. Doing so requires $2\ell$ $H$-applications. But the structural property of $H^2$ identified above means that, given $M$ and $Y$ one can compute $H^{2\ell}(H(M))$ using only one $H$-application:

$H(Y) = H(H^{2\ell}(M)) = H^{2\ell}(H(M))$. Moreover, the values computed along the first hash chain, namely the values $Y_i \leftarrow H^{2i}(M)$ and $Y_i' \leftarrow H^{2i}(H(M))$ for $0 \le i \le \ell$ are disjoint with overwhelming probability (when $\ell$ is not unreasonably large). Note that for chains of RO applications, attempting to cheaply compute such a second chain would not lead to disjoint chains. This demonstrates a way in which a RO and $H^2$ differ.

We exhibit a cryptographic setting, called *mutual proofs of work*, in which the highlighted structure of $H^2$ can be exploited. In mutual proofs of work, two parties prove to each other that they have computed some asserted amount of computational effort. This task is inspired by, and similar to, client puzzles [20, 21, 27, 28, 39] and puzzle auctions [41]. We give a protocol for mutual proofs of work whose computational task is computing hash chains. This protocol is secure when using a random oracle, but when using instead $H^2$ an attacker can cheat by abusing the structural properties discussed above.

Indifferentiability lower bound. The mutual proofs of work example already points to the surprising fact that $H^2$ does not "behave like" a RO. In fact, it does more, ruling out proofs of indifferentiability for $H^2$ with good bounds. (The existence of a tight proof of indifferentiability combined with the composition theorem of [33] would imply security for mutual proofs of work, yielding a contradiction.) However, we find that the example does not surface well why simulators must fail, and the subtlety of the issues here prompt further investigation. We therefore provide a direct negative result in the form of an indifferentiability distinguisher. We prove that should the distinguisher make $q_1, q_2$ queries to its two oracles, then for any simulator the indifferentiability advantage of the distinguisher is lower-bounded by $1 - (q_1 q_2)/q_S - q_S^2/2^n$. (This is slightly simpler than the real bound, see Section 3.3.) What this lower bound states is that the simulator must make very close to $\min\{q_1 q_2, 2^{n/2}\}$ queries to prevent this distinguisher's success. The result extends to structured underlying hash functions $H$ as well, for example should $H$ be MD-based.

To the best of our knowledge, our results are the first to show lower bounds on the number of queries an indifferentiability simulator must use. That a simulator must make a large number of queries hinders the utility of indifferentiability. When one uses the MRH composition theorem, the security of a scheme when using a monolithic RO must hold up to the number of queries the simulator makes. For example, in settings where one uses a hash function needing to be collision-resistant and attempts to conclude security via some (hypothetical) indifferentiability bound, our results indicate that the resulting security bound for the application can be at most $2^{n/4}$ instead of the expected $2^{n/2}$.

Upper bounds for second iterates. We have ruled out good upper bounds on indifferentiability, but the question remains whether weak bounds exist. We provide proofs of indifferentiability for $H^2$ that hold up to about $2^{n/4}$ distinguisher queries (our lower bounds rule out doing better) when $H$ is a RO. We provide some brief intuition about the proof. Consider an indifferentiability adversary making at most $q_1, q_2$ queries. The adversarial strategy of import is to compute long chains using the left oracle, and then try to "catch" the simulator in an inconsistency by querying it on a value at the end of the chain and, afterwards, filling in the intermediate values via further left and right queries. But the simulator can avoid being caught if it prepares long chains itself to help it answer queries consistently. Intuitively, as long as the simulator's chains are a bit longer than $q_1$ hops, then the adversary cannot build a longer chain itself (being restricted to at most $q_1$ queries) and will never win. The full proofs of these results are quite involved, and so we defer more discussion until the body. We are unaware of any indifferentiability proofs that requires this kind of nuanced strategy by the simulator.

## 1.2 HMAC with Arbitrary Keys

HMAC was introduced by Bellare, Canetti, and Krawczyk [5] to be used as a pseudorandom function or message authentication code. It uses an underlying hash function $H$; let $H$ have block size $d$ bits and output length $n$ bits. Computing a hash $\mathrm{HMAC}(K, M)$ works as follows [30]. If $|K| > d$ then redefine $K \leftarrow H(K)$. Let $K'$ be $K$ padded with sufficiently many zeros to get a $d$ bit string. Then $\mathrm{HMAC}(K, M) = H(K' \oplus \mathsf{opad} \,\|\, H(K' \oplus \mathsf{ipad} \,\|\, M))$ where $\mathsf{opad}$ and $\mathsf{ipad}$ are distinct $d$-bit constants. The original (provable security) analyses of HMAC focus on the setting that the key $K$ is honestly generated and secret [3, 5]. But what has happened is that HMAC's speed, ubiquity, and assumed security properties have lead it to be used in a wide variety of settings.

Of particular relevance are settings in which existing (or potential) proofs of security model HMAC as a keyed RO, a function that maps each key, message pair to an independent and uniform point. There are many examples of such settings. The HKDF scheme builds from HMAC a general-purpose key derivation function [31, 32] that uses as key a public, uniformly chosen salt. When used with a source of sufficiently high entropy, Krawczyk proves security using standard model techniques, but when not proves security assuming HMAC is a keyed RO [32]. PKCS#5 standardizes password-based key derivation functions that use HMAC with key being a (low-entropy) password [36]. Recent work provides the first proofs of security for the HMAC-based key derivation function, should HMAC be modeled as a keyed RO [9]. Ristenpart and Yilek [38], in the context of hedged cryptography [4], use HMAC in a setting whose cryptographic security models allow adversarially specified keys. Again, proofs model HMAC as a keyed RO.

As mentioned previously, we would expect a priori that one can show that HMAC is indifferentiable from a keyed RO even when the attacker can query arbitrary keys. Then one could apply the composition theorem of [33] to derive proofs of security for the settings just discussed.

Weak key pairs in HMAC. We are the first to observe that HMAC has weak key pairs. First, there exist $K \neq K'$ for which $\mathrm{HMAC}(K, M) = \mathrm{HMAC}(K', M)$. These pairs of keys arise because of HMAC's ambiguous encoding of differing-length keys. Trivial examples of such "colliding" keys include any $K, K'$ for which either $|K| < d$ and $K' = K \,\|\, 0^s$ (for any $1 \le s \le d - |K|$), or $|K| > d$ and $K' = H(K)$. Colliding keys enable an easy attack that distinguishes $\mathrm{HMAC}(\cdot, \cdot)$ from a random function $\mathcal{R}(\cdot, \cdot)$, which also violates the indifferentiability of HMAC. On the other hand, as long as $H$ is collision-resistant, two keys of the same length can never collide. Still, even if we restrict attention to (non-colliding) keys of a fixed length, there still exist weak key pairs, but of a different form that we term ambiguous. An example of an ambiguous key pair is $K, K'$ of length $d$ bits such that $K \oplus \mathsf{ipad} = K' \oplus \mathsf{opad}$. Because the second least significant bit of $\mathsf{ipad}$ and $\mathsf{opad}$ differ (see Section 4) and assuming $d > n - 2$, ambiguous key pairs of a fixed length $k$ only exist for $k \in \{d - 1, d\}$. The existence of ambiguous key pairs in HMAC leads to negative results like those given for $H^2$. In particular, we straightforwardly extend the $H^2$ distinguisher to give one that lower bounds the number of queries any indifferentiability simulator must make for HMAC.

Upper bounds for HMAC. Fortunately, it would seem that weak key pairs do not arise in typical applications. Using HMAC with keys of some fixed bit length smaller than $d - 1$ avoids weak key pairs. This holds for several applications, for example the recommendation with HKDF is to use $n$-bit uniformly chosen salts as HMAC keys. This motivates finding positive results for HMAC when one avoids the corner cases that allow attackers to exploit weak key pairs.

Indeed, as our main positive result, we prove that, should $H$ be a RO or an MD hash with ideal compression functions, *HMAC is indifferentiable from a keyed RO for all distinguishers that do not query weak key pairs.* Our result holds for the case that the keys queried are of length $d$ or less. This upper bound enjoys the best, birthday-bound level of concrete security possible (up to small constants), and provides the *first positive result about the indifferentiability of the HMAC construction.*

### 1.3 Discussion

The structural properties within $H^2$ and HMAC are, in theory, straightforward to avoid. Indeed, as mentioned above, Coron et al. [14] prove indifferentiable from a RO the construction $H^2(0^d \parallel M)$ where $H$ is MD using a compression function with block size $d$ bits and chaining value length $n \leq d$ bits. Analogously, our positive results about HMAC imply as a special case that $\mathrm{HMAC}(K, M)$, for any fixed constant $K$, is indifferentiable from a RO.

We emphasize that we are unaware of any deployed cryptographic application for which the use of $H^2$ or HMAC leads to a vulnerability. Still, our results show that future applications should, in particular, be careful when using HMAC with keys which are under partial control of the attacker. More importantly, our results demonstrate the importance of provable security in the design of hash functions (and elsewhere in cryptography), as opposed to the more common "attack-fix" cycle. For example, the hash-of-hash suggestion of Ferguson and Schneier [22] was motivated by preventing the extension attack. Unfortunately, in so doing they accidentally introduced a more subtle (although less dangerous) attack, which was not present on the original design.[1] Indeed, we discovered the subtlety of the problems within $H^2$ and HMAC, including our explicit attacks, only after attempting to prove indifferentiability of these constructions (with typical, good bounds). In contrast, the existing indifferentiability proofs of (seemingly) small modifications of these hash functions, such as $H^2(0^d \parallel M)$ [14], provably rule out these attacks.

### 1.4 Prior Work

There exists a large body of work showing hash functions are indifferentiable from a RO (c.f., [1, 7, 12–14, 18, 19, 26]), including analyses of variants of $H^2$ and HMAC. As mentioned, a construction called HMAC was analyzed in [14] but this construction is not HMAC as standardized. Krawczyk [32] suggests that the analysis of $H^2(0 \parallel M)$ extends to the case of HMAC, but does not offer proof.[2] HMAC has received much analysis in other contexts. Proofs of its security as a pseudorandom function under reasonable assumptions appear in [3, 5]. These rely on keys being uniform and secret, making the analyses inapplicable for other settings. Analysis of HMAC's security as a randomness extractor appear in [17, 23]. These results provide strong information theoretic guarantees that HMAC can be used as a key derivation function, but only in settings where the source has a relatively large amount of min-entropy. This requirement makes the analyses insufficient to argue security in many settings of practical importance. See [32] for further discussion.

## 2 Preliminaries

Notation and games. We denote the empty string by $\lambda$. If $|X| < |Y|$ then $X \oplus Y$ signifies that the $X$ is padded with $|Y| - |X|$ zeros first. For set $\mathcal{X}$ and value $x$, we write $\mathcal{X} \xleftarrow{\cup} x$ to denote $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$. For non-empty sets *Keys*, *Dom*, and *Rng* with $|Rng|$ finite, a random oracle $f\colon Keys \times Dom \to Rng$ is a function taken randomly from the space of all possible functions $Keys \times Dom \to Rng$. We will sometimes refer to random oracles as keyed when *Keys* is non-empty, whereas we omit the first parameter when $Keys = \emptyset$.

[1] We note the prescience of the proposers of $H^2$, who themselves suggested further analysis was needed [22].

[2] Fortunately, the HKDF application of [32] seems to avoid weak key pairs, and thus our positive results for HMAC appear to validate this claim [32] for this particular application.

We use code-based games [10] to formalize security notions and within our proofs. In the execution of a game $G$ with adversary $\mathcal{A}$, we denote by $G^{\mathcal{A}}$ the event that the game outputs true and by $\mathcal{A}^G \Rightarrow y$ the event that the adversary outputs $y$. Fixing some RAM model of computation, our convention is that the running time $\mathsf{Time}(\mathcal{A})$ of an algorithm $\mathcal{A}$ includes its code size. Queries are unit cost, and we will restrict attention to the absolute worst case running time which must hold regardless of queries are answered.

HASH FUNCTIONS. A *hash function* $H[P]\colon\ Keys \times Dom \to Rng$ is is a family of functions from $Dom$ to $Rng$, indexed by a set $Keys$, that possibly uses (black-box) access to an underlying primitive $P$ (e.g., a compression function). We call the hash function *keyed* if $Keys$ is non-empty, and key-less otherwise. (In the latter case, we omit the first parameter.) We assume that the number of applications of $P$ in computing $H[P](K,M)$ is the same for all $K,M$ with the same value of $|K|+|M|$. This allows us to define the *cost* of computing a hash function $H[P]$ on a key and message whose combined length is $\ell$, denoted $\mathsf{Cost}(H,\ell)$, as the number of calls to $P$ required to compute $H[P](K,M)$ for $K,M$ with $|K|+|M|=\ell$. For a keyed random oracle $\mathcal{R}\colon\ Keys \times Dom \to Rng$, we fix the convention that $\mathsf{Cost}(\mathcal{R},\ell)=1$ for any $\ell$ for which there exists a key $K \in Keys$ and message $M \in Dom$ such that $|K|+|M|=\ell$.

A *compression function* is a hash function for which $Dom=\{0,1\}^n \times \{0,1\}^d$ and $Rng=\{0,1\}^n$ for some numbers $n,d>0$. Our focus will be on keyless compression functions, meaning those of the form $f\colon\ \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$. Our results lift in a straightforward way to the dedicated-key setting [8]. The $\ell$-th iterate of $H[P]$ is denoted $H^\ell[P]$, and defined for $\ell>0$ by $H^\ell[P](X) = H[P](H[P](\cdots H[P](X))\cdots)$ where the number of applications of $H$ is $\ell$. We let $H^0[P](X)=X$. We will often write $H$ instead of $H[P]$ when the underlying primitive $P$ is clear or unimportant.

MERKLE-DAMGÅRD. Let $\mathsf{Pad}\colon\ \{0,1\}^{\leq L} \to (\{0,1\}^n)^+$ be an injective padding function. The one used in many of the hash functions within the SHA family outputs $M \parallel 10^r \parallel \langle |M| \rangle_{64}$ where $\langle |x| \rangle_{64}$ is the encoding of the length of $M$ as a 64-bit string and $r$ is the smallest number making the length a multiple of $d$. This makes $L=2^{64}-1$. The function $\mathrm{MD}[f]\colon\ (\{0,1\}^n)^+ \to \{0,1\}^n$ is defined as

$$\mathrm{MD}[f](M) = f(f(\cdots f(f(IV,M_1),M_2),\cdots),M_k)$$

where $|M|=kd$ and $M_1 \parallel \cdots \parallel M_k$. The function $\mathrm{SMD}[f]\colon\ \{0,1\}^{\leq L} \to \{0,1\}^n$ is defined as $\mathrm{SMD}[f](M) = \mathrm{MD}[f](\mathsf{Pad}(M))$.

INDIFFERENTIABILITY FROM A RO. Let $\mathcal{R}\colon\ Keys \times Dom \to Rng$ be a random oracle. Consider a hash construction $H[P]\colon\ Keys \times Dom \to Rng$ from an ideal primitive $P$. Let game $\mathrm{Real}_{H[P]}$ be the game whose main procedure runs an adversary $\mathcal{A}^{\mathsf{Func},\mathsf{Prim}}$ and returns the bit that $\mathcal{A}$ outputs. The procedure $\mathsf{Func}$ on input $K \in Keys$ and $M \in Dom$ returns $H[P](K,M)$. The procedure $\mathsf{Prim}$ on input $X$ returns $P(X)$. For a simulator $\mathcal{S}$, let game $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}$ be the game whose main procedure runs an adversary $\mathcal{A}^{\mathsf{Func},\mathsf{Prim}}$ and returns the bit that $\mathcal{A}$ outputs. The procedure $\mathsf{Func}$ on input $K \in Keys$ and $M \in Dom$ returns $\mathcal{R}(K,M)$. The procedure $\mathsf{Prim}$ on input $X$ returns $\mathcal{S}^{\mathcal{R}}(X)$. The indifferentiability advantage of $\mathcal{D}$ is defined as

$$\mathbf{Adv}^{\mathrm{indiff}}_{H[P],\mathcal{R},\mathcal{S}}(\mathcal{D}) = \Pr\left[\,\mathrm{Real}^{\mathcal{D}}_{H[P]} \Rightarrow y\,\right] - \Pr\left[\,\mathrm{Ideal}^{\mathcal{D}}_{\mathcal{R},\mathcal{S}} \Rightarrow y\,\right]\ .$$

We focus on simulators that must work for any adversary, though our negative results extend as well to the weaker setting in which the simulator can depend on the adversary. The total query cost $\sigma$ of an adversary $\mathcal{D}$ is the cumulative cost of all its $\mathsf{Func}$ queries plus $q_2$. (This makes $\sigma$ the total number of $P$ uses in game $\mathrm{Real}_{H[P]}$. In line with our worst-case conventions, this means the same maximums hold in $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}$ although here it does not translate to $P$ applications.)

We note that when $Keys$ is non-empty, indifferentiability here follows [8] and allows the distin-

| | |
|---|---|
| **main** $\mathrm{PrA}_{H,P,\mathcal{E}}$ | **procedure** $\mathsf{Prim}(u)$: |
| $\mathtt{V} \leftarrow \perp$ ; $\alpha \leftarrow \lambda$ | $v \leftarrow P(u)$ ; $\alpha \leftarrow \alpha \parallel (u,v)$ ; Ret $v$ |
| $(K,M) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim},\mathsf{Ext}}$ | |
| $z \leftarrow H[P](K,M)$ | **procedure** $\mathsf{Ext}(z)$: |
| Ret $((K,M) \neq \mathtt{V}[z] \wedge \mathtt{Q}[z] = 1)$ | $\mathtt{Q}[z] \leftarrow 1$ ; $\mathtt{V}[z] \leftarrow \mathcal{E}(z,\alpha)$ ; Ret $\mathtt{V}[z]$ |

Figure 1: The game for defining preimage awareness.

guisher to choose keys during an attack. This reflects the desire for a keyed hash function to be indistinguishable from a keyed random oracle for arbitrary uses of the key input.

PREIMAGE AWARENESS [19]. An *extractor* is a deterministic algorithm that takes as input a point $z$ and a string $\alpha$ (called the advice string) and returns a string or $\perp$. The game $\mathrm{PrA}_{H[P],\mathcal{E}}$ shown in Figure 1 defines the security experiment for preimage awareness. For a function $H$ using an ideal primitive $P$, an extractor $\mathcal{E}$, and an adversary $\mathcal{A}$ we define the advantage function

$$\mathbf{Adv}^{\mathrm{pra}}_{H[P],\mathcal{E}}(\mathcal{A}) = \Pr\left[\, \mathrm{PrA}^{\mathcal{A}}_{H[P],\mathcal{E}} \Rightarrow \mathsf{true} \,\right] .$$

# 3 Second Iterates and their Security

Our investigation begins with the second iterate of a hash function, meaning $H^2(M) = H(H(M))$ where $H\colon\ \mathit{Dom} \to \mathit{Rng}$ for sets $\mathit{Dom} \supseteq \mathit{Rng}$. For simplicity, let $\mathit{Rng} = \{0,1\}^n$ and assume that $H$ is itself modeled as a RO. Is $H^2$ good in the sense of being like a RO? Given that we are modeling $H$ as a RO, we would expect that the answer would be "yes". The truth is more involved. As we'll see in Section 4, similar subtleties exist in the case of the related HMAC construction.

We start with the following observations. When computing $H^2(M)$ for some $M$, we refer to the value $H(M)$ as an intermediate value. Then, we note that the value $Y = H^2(M)$ is in fact the intermediate value used when computing $H^2(X)$ for $X = H(M)$. Given $Y = H^2(M)$, then, one can compute $H^2(H(M))$ directly by computing $H(Y)$. That the hash value $Y$ is also the intermediate value used in computing the hash of another message is cause for concern: other hash function constructions that are indifferentiable from a RO (c.f., [2, 7, 8, 14, 26]) explicitly attempt to ensure that outputs are *not* intermediate values (with overwhelming probability over the randomness of the underlying idealized primitive). Moreover, prior constructions for which hash values are intermediate values have been shown to *not* be indifferentiable from a RO. For example Merkle-Damgård-based iterative hashes fall to extension attacks [14] for this reason. Unlike with Merkle-Damgård, however, it is not immediately clear how an attacker might abuse the structure of $H^2$.

## 3.1 Hash chains using Second Iterates

We turn our attention to hash chains, where potential issues arise. Hash chains, formed by repeatedly applying the hash function to some message, are used in a variety of settings such as password-based cryptography [36] and forward-secure pseudorandom number generators [11]. For a hash function $H$, we define a hash chain $Y = (Y_0, \ldots, Y_\ell)$ to be a sequence of $\ell + 1$ values where $Y_0$ is a message and $Y_i = H(Y_{i-1})$ for $1 \leq i \leq \ell$. Likewise when using $H^2$ a hash chain $Y = (Y_0, \ldots, Y_\ell)$ is a sequence of $\ell + 1$ values where $Y_0$ is a message and $Y_i = H^2(Y_{i-1})$ for $1 \leq i \leq \ell$. We refer to $Y_0$ as the *start* of the hash chain and $Y_\ell$ as the *end*. Two chains $Y, Y'$ are *non-overlapping* if no value in one chain occurs in the other, meaning $Y_i \neq Y'_j$ for all $0 \leq i \leq j \leq \ell$.

For any hash function and given the start and end of a hash chain $Y = (Y_0, \ldots, Y_\ell)$, one can readily

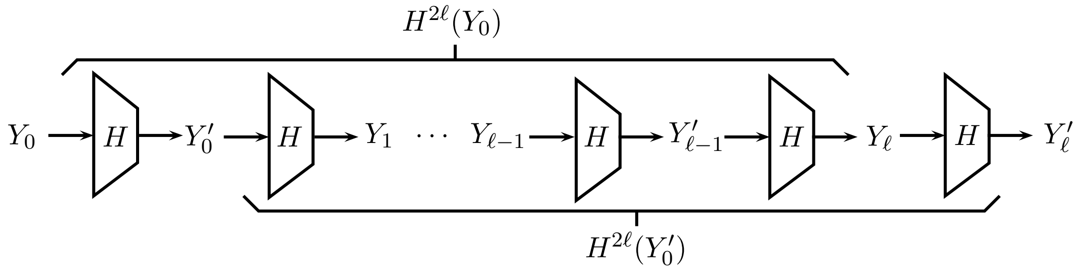


Figure 2: Diagram of two hash chains $Y = (Y_0, \ldots, Y_\ell)$ and $Y' = (Y'_0, \ldots, Y'_\ell)$ for hash function $H^2$.

compute the start and end of a new chain with just two hash calculations. That is, set $Y'_0 \leftarrow H(Y_0)$ and $Y'_\ell \leftarrow H(Y_\ell)$. However, the chain $Y' = (Y'_0, \ldots, Y'_\ell)$ and the chain $Y$ overlap. For good hash functions (i.e., ones that behave like a RO) computing the start and end of a non-overlapping chain given the start and end of a chain $Y_0, Y_\ell$ requires at least $\ell$ hash computations (assuming $\ell \ll 2^{n/2}$).

Now consider $H^2$. Given the start and end of a chain $Y = (Y_0, \ldots, Y_\ell)$, one can readily compute a non-overlapping chain $Y' = (Y'_0, \ldots, Y'_\ell)$ using just two hash computations instead of the expected $2\ell$ computations. Namely, let $Y'_0 \leftarrow H(Y_0)$ and $Y'_\ell \leftarrow H(Y_\ell)$. Then these are the start and end of the chain $Y' = (Y'_0, \ldots, Y'_\ell)$ because

$$H^{2\ell}(Y'_0) = H^{2\ell}(H(Y_0)) = H(H^{2\ell}(Y_0))$$

which we call the chain-shift property of $H^2$. Moreover, assuming $H$ is itself a RO outputing $n$-bit strings, the two chains $Y, Y'$ do not overlap with probability at least $1-(2\ell+2)^2/2^n$. Figure 2 provides a pictoral diagram of the two chains $Y$ and $Y'$.

The above discussion points out a way in which $H^2$ and a RO differ. We now formalize this difference via a toy security game. Game $\mathrm{CHAIN}_{H[P],n,\ell}$ is shown in Figure 3. It tasks an attacker with computation of a hash chain of length $\ell$ for a hash function $H[P]$. The attacker is additionally given the start and end of an honestly generated hash chain, and the attacker can make at most $\ell \cdot \mathsf{Cost}(H,n) - 1$ queries to the primitive underlying the hash construction — one less query than that needed to compute a new hash chain directly. The attacker succeeds if it can nevertheless compute a the start and end of a hash chain that does not overlap with the honestly generated one. We define the advantage of an $\mathrm{CHAIN}_{H[P],n,\ell}$-adversary $\mathcal{A}$ by

$$\mathbf{Adv}^{\mathrm{chain}}_{H[P],n,\ell}(\mathcal{A}) = \Pr\left[\ \mathrm{CHAIN}^{\mathcal{A}}_{H[P],n,\ell} \Rightarrow \mathsf{true}\ \right] .$$

We consider below two constructions:

- The second iterate $H^2[P]$ defined by computing $P(P(M))$ and where $P\colon\ \mathit{Dom} \to \{0,1\}^n$ that is a random oracle. To win, the attacker must somehow compute a length $\ell$ hash chain for $H^2[P]$ using at most $2\ell - 1$ queries to $P$.
- The hash function is a random oracle, formally $H[P]$ is defined by $P(M)$ where $P\colon\ \mathit{Dom} \to \{0,1\}^n$ is a RO. To win, the attacker must somehow compute a length $\ell$ hash chain for $H[P]$ using at most $\ell - 1$ queries to $P$.

Chain security of a RO. We start with the latter bullet point: We show that no attacker can achieve good advantage in the $\mathrm{CHAIN}_{H,P,n,\ell}$ game for $H[P](M) = P(M)$ when $\ell \ll 2^{n/2}$.

**Claim 3.1** Let $H[P]$ be the hash that applies a RO $P\colon\ \mathit{Dom} \to \{0,1\}^n$ to each input with $\mathit{Dom} \supseteq \{0,1\}^n$ and let $\ell \in [1 \,..\, 2^n/6]$. Then for any $\mathcal{A}$ it holds that $\mathbf{Adv}^{\mathrm{chain}}_{H,P,n,\ell}(\mathcal{A}) \le \frac{(3\ell)^2}{2^n}$. $\square$

| | |
|---|---|
| **main** $\text{CHAIN}_{H,P,n,\ell}$: | **procedure** $\mathsf{Prim}(M)$: |
| $i \leftarrow 0$ | $i \leftarrow i+1$ |
| $Y_0 \leftarrow\!\!\$\ \{0,1\}^n$ ; For $i=1$ to $\ell$ do $Y_i \leftarrow H[P](Y_{i-1})$ | If $i \geq \ell \cdot \mathsf{Cost}(H,n)$ then Ret $\perp$ |
| $(Y_0^*, Y_\ell^*) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(Y_0, Y_\ell)$ | Ret $P(M)$ |
| $Y_0' \leftarrow Y_0^*$ ; For $i=1$ to $\ell$ do $Y_i' \leftarrow H[P](Y_{i-1}')$ | |
| $\mathcal{Y} \leftarrow \{Y_0, \ldots, Y_\ell\}$ | |
| $\mathcal{Y}' \leftarrow \{Y_0', \ldots, Y_\ell'\}$ | |
| If $(Y_\ell' = Y_\ell^*) \wedge ((\mathcal{Y} \cap \mathcal{Y}') = \emptyset)$ then Ret true | |
| Ret false | |

Figure 3: The chain-making game.

**Proof:** We can assume that $\mathcal{A}$ makes exactly $\ell - 1$ queries to Prim. Let $\tilde{P}\colon\ Dom \to \{0,1\}^n$ map each input to an output selected uniformly without replacement. Then a standard argument shows that

$$\mathbf{Adv}^{\text{chain}}_{H,P,n,\ell}(\mathcal{A}) \leq \mathbf{Adv}^{\text{chain}}_{H,\tilde{P},n,\ell}(\mathcal{A}) + \frac{(3\ell-1)^2}{2^n} . \tag{1}$$

To win the game gainst $H[\tilde{P}]$, the adversary $\mathcal{A}$ must output $Y_0^*, Y_\ell^*$ such that the associated chain $Y_0', \ldots, Y_\ell'$ is non-overlapping with $Y_0, \ldots, Y_\ell$ and with $Y_0' = Y_0^*$ and $Y_\ell' = Y_\ell^*$. But $\mathcal{A}$ observes $\ell - 1$ of the $\ell$ values $Y_0', \ldots, Y_\ell'$. Let $i \in [1\mathinner{..}\ell]$ be the index such that $Y_i'$ was not returned to $\mathcal{A}$ by Prim. First consider the case that $i = \ell$. Then, $Y_\ell'$ is a fresh choice made after $\mathcal{A}$ finishes execution, and so $Y_\ell^* = Y_\ell'$ with probability $1/(2^n - (3\ell-1))$. (Since we are disallowing collisions, $Y_\ell'$ is uniformly selected from a set of size $2^n - (3\ell - 1)$.) Now consider the case that $i < \ell$. Then, for the chain to be completed, the value $Y_i'$, chosen after $\mathcal{A}$ finishes executing, must equal the value $M$ queried by $\mathcal{A}$ that had response $Y_{i+1}'$. But this occurs, again, with probability at most $1/(2^n - (3\ell-1))$. Since we restricted attention to $1 \leq \ell \leq 2^n/6$, it holds that $\mathbf{Adv}^{\text{chain}}_{H,\tilde{P},n,\ell}(\mathcal{A}) \leq 2/2^n$. Substituting into (1) above yields the claim. ▮

Attack against $H^2$. Now let the hash function be $H^2[P] = P(P(M))$ for $P\colon\ Dom \to \{0,1\}^n$ a RO. We give an adversary $\mathcal{A}$ that wins the $\text{CHAIN}_{H^2[P],n,\ell}$ game with probability $1 - \ell^2/2^n$. Let $\mathcal{A}$ work as follows. Upon execution with inputs $Y_0, Y_\ell$, it queries $Y_0^* \leftarrow \mathsf{Prim}(Y_0)$ and $Y_\ell^* \leftarrow \mathsf{Prim}(Y_\ell)$. It then returns $(Y_0^*, Y_\ell^*)$. Then, letting $Y_0' = Y_0^*$ and $Y_1', \ldots, Y_\ell'$ be the values computed in the main procedure of the $\text{CHAIN}_{H^2[P],n,\ell}$ game, we see that the chain-shift property of $H^2$ means that $Y_\ell' = Y_\ell^*$. Moreover, no element in $Y_0', \ldots, Y_\ell'$ collides with an element in $Y_0, \ldots, Y_\ell$ with probability $1 - \ell^2/2^n$, this probability being over the random coins used by $P$. The adversary therefore achieves the stated advantage.

Discussion. All the above exhibits a way in which $H^2$ fails to behave like a RO. Moreover, the attack against $H^2$ generalizes to other hash function constructions that are second iterates, for example when using the second iterate of the Merkle-Damgård construction [16, 34].

But what does it imply about indifferentiability? Recall that the composition theorem of Maurer et al. [33], as discussed further by Ristenpart et al. [37], states —informally speaking— that a cryptosystem secure relative to any single-stage[3] game in the ROM will remain secure when replacing the RO with a hash construction that is indifferentiable from a RO. We have above given a game in which no attacker succeeds in the ROM, but there exists an adversary that succeeds when the RO is replaced with $H^2$. This might seem to directly rule out $H^2$ being indifferentiable from a RO. But we must

[3] The security games we consider fall into this category, which just mandates that the adversary maintains state throughout the entire experiment. See [37] for further discussion.

be careful. The gap between security and insecurity above is conditioned upon the limited number of queries, and such limitations must be carefully handled when indifferentiability is considered. In particular, the results above only would lead to contradicting a positive indifferentiability result using a simulator that does not make so many queries that the bound in Claim 3.1 becomes close to one.

We will tackle these subtleties head-on in Section 3.3, by giving a distinguisher that differentiates $H^2$ from a RO for any simulator that does not make sufficiently many queries. Before that, we first explore the implications of the surfaced structural property of $H^2$: Might it lead to vulnerabilities in applications?

## 3.2 An Example (Vulnerable) Application: Mutual Proofs of Work

In the last section we saw that the second iterate fails to behave like a RO in the context of hash chains. But the security game detailed in the last section may seem far removed from real protocols. For example, it's not clear where an attacker would be tasked with computing hash chains in a setting where it, too, was given an example hash chain. We suggest that just such a setting could arise in protocols in which parties want to assert to each other, in a verifiable way, that they performed some amount of computation. Such a setting could arise when parties must (provably) compare assertions of computational power, as when using cryptographic puzzles [20, 21, 27, 28, 39, 41]. Or this might work when trying to verifiably calibrate differing computational speeds of the two parties' computers. We refer to this task as a *mutual proof of work*.

MUTUAL PROOFS-OF-WORK. For the sake of brevity, we present an example hash-chain-based protocol and dispense with a more general treatment of mutual proofs of work. Consider the two-party protocol shown in the left diagram of Figure 4. Each party initially chooses a random nonce and sends it to the other. Then, each party computes a hash chain of some length —chosen by the computing party— starting with the nonce chosen by the other party, and sends the chain's output along with the chain's length to the other party. At this point, both parties have given a witness that they performed a certain amount of work. So now, each party checks the other's asserted computation, determining if the received value is the value resulting from chaining together the indicated number of hash applications and checking that the hash chains used by each party are non-overlapping. Note that unlike puzzles, which require fast verification, here the verification step is as costly as puzzle solution.

The goal of the protocol is to ensure that the other party did compute exactly their declared number of iterations. Slight changes to the protocol would lead to easy ways of cheating. For example, if during verification the parties did not check that the chains are non-overlapping, then $\mathcal{P}_2$ can easily cheat by choosing $X_1$ so that it can reuse a portion of the chain computed by $\mathcal{P}_1$

Security would be achieved should no cheating party succeed at convincing an honest party using less than $\ell_1$ (resp. $\ell_2$) work to compute $Y_1$ (resp. $Y_2$). The game $\mathrm{POW}_{H[P],n,\ell_1}$ formalizes this security goal for a cheating $\mathcal{P}_2$; see the right portion of Figure 4. We let $\mathbf{Adv}^{\mathrm{pow}}_{H[P],n,\ell_1}(\mathcal{A}) = \Pr\left[\,\mathrm{POW}^{\mathcal{A}}_{H[P],n,\ell_1}\,\right]$. Note that the adversary $\mathcal{A}$ only wins should it make $q < \ell_2 \cdot \mathsf{Cost}(H,n)$ queries, where $\ell_2$ is the value it declared and $\mathsf{Cost}(H)$ is the cost of computing $H$. Again we will consider both the hash function $H[P](M) = P(M)$ that just applies a RO $P$ and also $H^2[P](M) = P(P(M))$, the second iterate of a RO. In the former case the can make only $\ell_2 - 1$ queries and in the latter case $2\ell_2 - 1$.

When $H[P](M) = P(M)$, no adversary making $q < \ell_2$ queries to $\mathsf{Prim}$ can win the $\mathrm{POW}_{H[P],n,\ell_1}$ game with high advantage. Intuitively, the reason is that, despite being given $X_1$ and $Y_1$ where $Y_1 = P^{\ell_1}(X_1)$, a successful attacker must still compute a full $\ell_2$-length chain and this requires $\ell_2$ calls to $P$. A treatment of this follows closely Claim 3.1 and its proof, and so we omit the details. Intuitively, the reason is that, despite being given $X_1$ and $Y_1$ where $Y_1 = P^{\ell_1}(X_1)$, a successful attacker

| $\mathcal{P}_1$ | | $\mathcal{P}_2$ |
|---|---|---|
| $X_2 \leftarrow\!\!\$\ \{0,1\}^n$ | $\xrightarrow{X_2}$ | $X_1 \leftarrow\!\!\$\ \{0,1\}^n$ |
| | $\xleftarrow{X_1}$ | |
| $Y_1 \leftarrow H^{\ell_1}(X_1)$ | $\xrightarrow{\ell_1, Y_1}$ | $Y_2 \leftarrow H^{\ell_2}(X_2)$ |
| | $\xleftarrow{\ell_2, Y_2}$ | |
| $\hat{\mathcal{Y}}_1 \leftarrow \{H^i(X_1) \mid 0 \le i \le \ell_1\}$ | | $\hat{\mathcal{Y}}_1 \leftarrow \{H^i(X_1) \mid 0 \le i \le \ell_1\}$ |
| $\hat{\mathcal{Y}}_2 \leftarrow \{H^i(X_2) \mid 0 \le i \le \ell_2\}$ | | $\hat{\mathcal{Y}}_2 \leftarrow \{H^i(X_2) \mid 0 \le i \le \ell_2\}$ |
| $Y_2' \leftarrow H^{\ell_2}(X_2)$ | | $Y_1' \leftarrow H^{\ell_1}(X_1)$ |
| Ret $(Y_2' = Y_2) \wedge (\hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$ | | Ret $(Y_1' = Y_1) \wedge (\hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$ |

**main** $\mathrm{POW}_{H[P],n,\ell_1}$:
$X_2 \leftarrow\!\!\$\ \{0,1\}^n$
$X_1 \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(X_2)$
$Y_1 \leftarrow H^{\ell_1}[P](X_1)$
$(\ell_2, Y_2) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(\ell_1, Y_1)$
$\hat{\mathcal{Y}}_1 \leftarrow \{H^i[P](X_1) \mid 0 \le i \le \ell_1\}$
$\hat{\mathcal{Y}}_2 \leftarrow \{H^i[P](X_2) \mid 0 \le i \le \ell_2\}$
$Y_2' \leftarrow H^{\ell_2}[P](X_2)$
If $q \ge \ell_2 \cdot \mathsf{Cost}(H, n)$ then
  Ret false
Ret $(Y_2' = Y_2 \wedge \hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$

**subroutine** $\mathsf{Prim}(u)$
$q \leftarrow q + 1$ ; Ret $P(u)$

Figure 4: Example protocol **(left)** and adversarial $\mathcal{P}_2$ security game **(right)** for mutual proofs of work.

must still compute a full $\ell_2$-length chain and this requires $\ell_2$ calls to $P$.

Attack against any second iterate. Now let us analyze this protocol's security when we use as hash function $H^2[P] = P(P(M))$ for a RO $P\colon \mathit{Dom} \to \mathit{Rng}$ with $\mathit{Rng} \subseteq \mathit{Dom}$. We can abuse the chain-shift property of $H^2$ in order to win the $\mathrm{POW}_{H^2,P,n,\ell_1}$ game for any $n > 0$ and $\ell_1 > 2$. Our adversary $\mathcal{A}$ works as follows. It receives $X_2$ and then chooses it's nonce as $X_1 \leftarrow \mathsf{Prim}(X_2)$. When it later receives $Y_1 = P^{2\ell_1}(X_1)$, the adversary proceeds by setting $\ell_2 = \ell_1 + 1$ and setting $Y_2 \leftarrow \mathsf{Prim}(Y_1)$. Then by the chain-shift property we have that

$$Y_2 = P(Y_1) = P(P^{2\ell_1}(X_1)) = P(P^{2\ell_1}(P(X_2))) = P^{2\ell_1+2}(X_2)) = P^{2\ell_2}(X_2)\ .$$

The two chains will be non-overlapping with high probability (over the coins used by $P$). Finally, $\mathcal{A}$ makes only 2 queries to $\mathsf{Prim}$, so the requirement that $q < 2\ell_2$ is met whenever $\ell_1 > 1$.

Discussion. As far as we are aware, mutual proofs of work have not before been considered — the concept may indeed be of independent interest. A full treatment is beyond the scope of this work. We also note that, of course, it is easy to modify the protocols using $H^2$ to be secure. Providing secure constructions was not our goal, rather we wanted to show protocols which are insecure using $H^2$ but secure when $H^2$ is replaced by a monolothic RO. This illustrates how, hypothetically, the structure of $H^2$ could give rise to subtle vulnerabilities in an application.

### 3.3 An Indifferentiability Distinguisher for any Second Iterate

In this section we prove that *any* indifferentiability proof for the double iterate $H^2$ is subject to inherent quantitative limitations. Recall that indifferentiability asks for a simulator $\mathcal{S}$ such that no adversary can distinguish between the pair of oracles $H^2[P], P$ and $\mathcal{R}, \mathcal{S}$ where $P$ is some underlying ideal primitive and $\mathcal{R}$ is a RO with the same domain and range as $H^2$. The simulator can make queries to $\mathcal{R}$ to help it in its simulation of $P$. Concretely, building on the ideas behind the above attacks in the context of hash chains, we show that in order to withstand a differentiating attack with $q$ queries, any simulator for $H^2[P]$, for *any* hash construction $H[P]$ with output length $n$, must issue *at least* $\Omega(\min\{q^2, 2^{n/2}\})$ queries to the RO $\mathcal{R}$. As we explain below, such a lower bound severely limits the concrete security level which can be inferred by using the composition theorem for indifferentiability, effectively neutralizing the benefits of using indifferentiability in the first place.

THE DISTINGUISHER. In the following, we let $H = H[P]$ be an *arbitrary* hash function with $n$-bit outputs relying on a primitive $P$, such as a fixed input-length random oracle or an ideal cipher. We are therefore addressing an arbitrary second iterate, and not focusing on some particular ideal primitive $P$ (such as a RO as in previous sections) or construction $H$. Indeed, $H$ could equally well be Merkle-Damgård and $P$ an ideal compression function, or $H$ could be any number of indifferentiable hash constructions using appropriate ideal primitive $P$.

Recall that Func and Prim are the oracles associated with construction and primitive queries to $H^2 = H^2[P]$ and $P$, respectively. Let $w, \ell$ be parameters (for now, think for convenience of $w = \ell$). The attacker $\mathcal{D}_{w,\ell}$ starts by issuing $\ell$ queries to Func to compute a *chain* of $n$-bit values $(x_0, x_1, \ldots, x_\ell)$ where $x_i = H^2(x_{i-1})$ and $x_0$ is a random $n$-bit string. Then, it also picks a random index $j \in [1\,..\,w]$, and creates a list of $n$-bit strings $\mathbf{u}[1], \ldots, \mathbf{u}[w]$ with $\mathbf{u}[j] = x_\ell$, and all remaining $\mathbf{u}[i]$ for $i \neq j$ are chosen uniformly and independently. Then, for all $i \in [1\,..\,w]$, the distinguisher $\mathcal{D}_{w,\ell}$ proceeds in asking all Prim queries in order to compute $\mathbf{v}[i] = H(\mathbf{u}[i])$. Subsequently, the attacker compute $y_0 = H(x_0)$ via Prim queries, and also computes the chain $(y_0, y_1, \ldots, y_\ell)$ such that $y_i = H^2(y_{i-1})$ by making $\ell$ Func queries. Finally, it decides to output 1 if and only if $y_\ell = \mathbf{v}[j]$ and $x_\ell$ as well as $\mathbf{v}[i]$ for $i \neq j$ are not in $\{y_0, y_1, \ldots, y_\ell\}$. The attacker $\mathcal{D}_{w,\ell}$ therefore issues a total of $2\ell$ Func queries and $(2w + 1) \cdot \mathsf{Cost}(H, n)$ Prim queries.

In the real-world experiment, the distinguisher $\mathcal{D}_{w,\ell}$ outputs 1 with very high probability, as the condition $y_\ell = \mathbf{v}[j]$ *always* holds by the chain-shifting property of $H^2$. In fact, the only reason for $\mathcal{D}$ outputting 0 is that one of $x_\ell$ and $\mathbf{v}[i]$ for $i \neq j$ incidentally happens to be in $\{y_0, y_1, \ldots, y_\ell\}$. The (typically small) probability that this occurs obviously depends on the particular construction $H[P]$ at hand; it is thus convenient to define the shorthand

$$p(H, w, \ell) = \Pr\left[\, \{x_\ell, H(U_1), \ldots, H(U_{w-1})\} \cap \{y_0, y_1, \ldots, y_\ell\} \neq \emptyset \,\right] \ ,$$

where $x_0, y_0, x_1, \ldots, y_{\ell-1}, x_\ell, y_\ell$ are the intermediate value of a chain of $2\ell + 1$ consecutive evaluations of $H[P]$ starting at a random $n$-bit string $x_0$, and $U_1, \ldots, U_{w-1}$ are further independent random $n$-bit values. In Appendix A we prove that for $H[P] = P = \mathcal{R}$ for a random oracle $\mathcal{R} : \{0,1\}^* \to \{0,1\}^n$ we have $p(H, w, \ell) = \Theta((w\ell + \ell^2)/2^n)$. Similar reasoning can be applied to essentially all relevant constructions.

In contrast, in the ideal-world experiment, we expect the simulator to be completely ignorant about the choice of $j$ as long as it does not learn $x_0$, and in particular it does not know $j$ while answering the Prim queries associated with the evaluations of $H(\mathbf{u}[i])$. Consequently, the condition required for $\mathcal{D}_{w,\ell}$ to output 1 appears to force the simulator, for all $i \in [1\,..\,w]$, to prepare a distinct chain of $\ell$ consecutive $\mathcal{R}$ evaluations ending in $\mathbf{v}[i]$, hence requiring $w \cdot \ell$ random oracle queries.

The following theorem quantifies the advantage achieved by the above distinguisher $\mathcal{D}_{w,\ell}$ in differentiating against any simulator for the construction $H[P]$. Its proof is given in Appendix B.

**Theorem 3.2 [Attack against $H^2$]** Let $H[P]$ be an arbitrary hash construction with $n$-bit outputs, calling a primitive $P$, and let $\mathcal{R} : \{0,1\}^* \to \{0,1\}^n$ be a random oracle. For all integer parameters $w, \ell \geq 1$, there exists an adversary $\mathcal{D}_{w,\ell}$ making $2\ell$ Func-queries and $(w + 1) \cdot \mathsf{Cost}(H, n)$ Prim-queries such that for all simulators $\mathcal{S}$,

$$\mathbf{Adv}^{\mathrm{indiff}}_{H^2[P],\mathcal{R},\mathcal{S}}(\mathcal{D}_{w,\ell}) \geq 1 - p(H, w, \ell) - \frac{5\ell^2}{2^{n+1}} - \frac{q_{\mathcal{S}}\ell}{2^n} - \frac{q_{\mathcal{S}}^2}{2^n} - \frac{q_{\mathcal{S}}}{w \cdot \ell} - \frac{1}{w} \ ,$$

where $q_{\mathcal{S}}$ is the overall number of $\mathcal{R}$ queries by $\mathcal{S}$ when replying to $\mathcal{D}_{w,\ell}$'s Prim queries. ∎

DISCUSSION. We now elaborate on Theorem 3.2. If we consider the distinguisher $\mathcal{D}_{w,\ell}$ from Theorem 3.2, we observe that by the advantage lower bound in the theorem statement, if $\ell, w \ll 2^{n/4}$ and consequently $p(H, w, \ell) \approx 0$, the number of queries made by the simulator, denoted $q_{\mathcal{S}} = q_{\mathcal{S}}(2\ell, w + 1)$

must satisfy $q_{\mathcal{S}} = \Omega(w \cdot \ell) = \Omega(q_1 \cdot q_2)$ to ensure a sufficiently small indifferentiability advantage. This in particular means that in the case where both $q_1$ and $q_2$ are large, the simulator must make a *quadratic* effort to prevent the attacker from distinguishing. Below, in Theorem 3.3, we show that this simulation effort is essentially optimal.

In many scenarios, this quadratic lower bound happens to be a problem, as we now illustrate. As a concrete example, let $\mathcal{SS} = (\mathsf{key}, \mathsf{sign}, \mathsf{ver})$ be an arbitrary signature scheme signing $n$ bits messages, and let $\widetilde{\mathcal{SS}}[\mathcal{R}] = (\widetilde{\mathsf{key}}^{\mathcal{R}}, \widetilde{\mathsf{sign}}^{\mathcal{R}}, \widetilde{\mathsf{ver}}^{\mathcal{R}})$ for $\mathcal{R} : \{0,1\}^* \to \{0,1\}^n$ be the scheme obtained via the hash-then-sign paradigm such that $\widetilde{\mathsf{sign}}^{\mathcal{R}}(sk, m) = \mathsf{sign}(sk, \mathcal{R}(m))$. It is well known that for an adversary $\mathcal{B}$ making $q_{\mathsf{sign}}$ signing and $q_{\mathcal{R}}$ random oracle queries, there exists an adversary $\mathcal{C}$ making $q_{\mathsf{sign}}$ signing queries such that

$$\mathbf{Adv}^{\text{uf-cma}}_{\widetilde{\mathcal{SS}}[\mathcal{R}]}(\mathcal{B}^{\mathcal{R}}) \leq \frac{(q_{\mathsf{sign}} + q_{\mathcal{R}})^2}{2^n} + \mathbf{Adv}^{\text{uf-cma}}_{\mathcal{SS}}(\mathcal{C}) \;, \tag{2}$$

where $\mathbf{Adv}^{\text{uf-cma}}_{\widetilde{\mathcal{SS}}[\mathcal{R}]}(\mathcal{B}^{\mathcal{R}})$ and $\mathbf{Adv}^{\text{uf-cma}}_{\mathcal{SS}}(\mathcal{C})$ denote the respective advantages in the standard uf-cma game for security of signature schemes (with and without a random oracle, respectively). This in particular means that $\widetilde{\mathcal{SS}}$ is secure for $q_{\mathsf{sign}}$ and $q_{\mathcal{R}}$ as large as $\Theta(2^{n/2})$, provided $\mathcal{SS}$ is secure for $q_{\mathsf{sign}}$ signing queries. However, let us now replace $\mathcal{R}$ by $H^2[P]$ for an arbitrary construction $H = H[P]$. Then, for all adversaries $\mathcal{A}$ making $q_P$ queries to $P$ and $q_{\mathsf{sign}}$ signing queries, we can combine the concrete version of the MRH composition theorem proven in [37] and (2) to infer that there exists an adversary $\mathcal{C}$ and a distinguisher $\mathcal{D}$ such that

$$\mathbf{Adv}^{\text{uf-cma}}_{\widetilde{\mathcal{SS}}[H^2[P]]}(\mathcal{A}^{P}) \leq \Theta\left(\frac{(q_{\mathsf{sign}} \cdot q_P)^2}{2^n}\right) + \mathbf{Adv}^{\text{uf-cma}}_{\mathcal{SS}}(\mathcal{C}) + \mathbf{Adv}^{\text{indiff}}_{H^2[P],\mathcal{R},\mathcal{S}}(\mathcal{D}) \;,$$

where $\mathcal{C}$ makes $q_{\mathsf{sign}}$ signing queries . Note that even if the term $\mathbf{Adv}^{\text{indiff}}_{H^2[P],\mathcal{R},\mathcal{S}}(\mathcal{D})$ is really small, this new bound can only ensure security for the resulting signature scheme as long as $q_{\mathsf{sign}} \cdot q_P = \Theta(2^{n/2})$, i.e., if $q_{\mathsf{sign}} = q_P$, we only get security up to $\Theta(2^{n/4})$ queries, a remarkable loss with respect to the security bound in the random oracle model.

We note that of course this does *not* mean that $H^2[P]$ for a concrete $H$ and $P$ is unsuitable for a certain application, such as hash-then-sign. In fact, $H^2[P]$ may well be optimally collision resistant. However, our result shows that a sufficiently strong security level cannot be inferred from *any* indifferentiability statement via the composition theorem, taking us back to a direct ad-hoc analysis and completely loosing the one main advantage of having indifferentiability in the first place.

### 3.4 Indifferentiability Upper Bound for a Second Iterate

Our negative results do not rule out positive results completely: there could be indifferentiability upper bounds, though for simulators that make around $\mathcal{O}(q^2)$ queries. Ideally, we would like upper bounds that match closely the lower bounds given in prior sections. We do so for the special case of $H^2[g](M) = g(g(M))$ for $g\colon \{0,1\}^n \to \{0,1\}^n$ being a RO. We have the following theorem.

**Theorem 3.3** Let $q_1, q_2 \geq 0$ and $N = 2^n$. Let $g : \{0,1\}^n \to \{0,1\}^n$ and $\mathcal{R} : \{0,1\}^n \to \{0,1\}^n$ be uniform random functions. Then there exists a simulator $\mathcal{S}$ such that

$$\mathbf{Adv}^{\text{indiff}}_{G[g],\mathcal{R},\mathcal{S}}(\mathcal{D}) \leq \frac{2((4q_1+3)q_2 + 2q_1)^2}{N} + \frac{2((4q_1+3)q_2+2q_1)(q_1+q_2)}{(N - 2q_2 - 2q_1)}$$

for any adversary $\mathcal{D}$ making at most $q_1$ queries to its left oracle and at most $q_2$ queries to its right oracle. Moreover, for each query answer that it computes, $\mathcal{S}$ makes at most $3q_1 + 1$ queries to RO and runs in time $O(q_1)$. $\square$

| | | |
|---|---|---|
| **procedure** OnRightQuery($x$): | **subroutine** FillInRungs($x, y$) | **subroutine** MakeLadder($x$) |
| $x_0 \leftarrow x$ | $x_0 \leftarrow x, x_1 \leftarrow y$ | $s_{-q_1} \leftarrow\$ \{0,1\}^n$ |
| For $i = 0$ to $q_1$ | SetTable($\mathtt{g}, x_0, x_1$) | For $i = -q_1$ to $q_1 - 1$ |
|   If $\mathtt{g}[x_i] \neq \perp$ then | For $i = 1$ to $2q_1 + 1$ |   SetTable($\mathtt{G}, s_i, \mathrm{RO}(s_i)$) |
|     $y \leftarrow \mathtt{G}^{-i}[\mathtt{g}[x_i]]$ |   SetTable($\mathtt{G}, x_{i-1}, \mathrm{RO}(x_{i-1})$) |   $s_{i+1} \leftarrow \mathtt{G}[s_i]$ |
|     If $y = \perp$ then Abort |   $x_{i+1} \leftarrow \mathtt{G}[x_{i-1}]$ | FillInRungs($x, s_0$) |
|     FillInRungs($x, y$) |   SetTable($\mathtt{g}, x_i, x_{i+1}$) | **subroutine** SetTable($\mathtt{T}, x, y$) |
|     Ret $y$ | | If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret |
|   SetTable($\mathtt{G}, x_i, \mathrm{RO}(x_i)$) | | $\mathtt{T}[x] = y$ |
|   $x_{i+1} \leftarrow \mathtt{G}[x_i]$ | | $\mathtt{T}^{-1}[y] = x$ |
| MakeLadder($x$) | | |
| Ret $\mathtt{g}[x]$ | | |

Figure 5: Simulator $\mathcal{S}$ used in the proof of Theorem 3.3.

We note that the security bound of Theorem 3.3 is approximately $(q_1q_2)^2/N$, implying that Theorem 3.3 guarantees security up to $q_1q_2 \approx 2^{n/2}$. For example, $G[g]$ would be indifferentiable to an attacker making $o(2^{\frac{1}{3}n})$ left oracle queries and $o(2^{\frac{1}{6}n})$ right oracle queries, or $O(1)$ left oracle queries and $o(2^{n/2})$ right oracle queries, or, oppositely, $o(2^{n/2})$ left oracle queries $O(1)$ right oracle queries, and so on. If all that is known is the attacker's total number of queries, then all one can say is that security is guaranteed up to $\approx 2^{n/4}$ queries.

Proof sketch. The simulator $\mathcal{S}$ referred to in Theorem 3.3 is implemented by the procedure OnRightQuery($\cdot$) shown in Figure 5. Here we give some intuition about $\mathcal{S}$ —what it does and why— along with a brief overview of the full proof, which appears in Appendix D.

We use the notation $\mathrm{RO}(x) \to y$ to indicate that the left oracle (in the ideal world) returns $y$ when queried at $x$. Consider an attacker $\mathcal{D}$ that starts by making a sequence of two queries $\mathrm{RO}(t_{-2}) \to t_{-1}$, $\mathrm{RO}(t_{-1}) \to t_0$ for a randomly chosen $t_{-2} \in \{0,1\}^n$, and then queries the simulator at $t_0$. At this point the simulator has no knowledge of any of the adversary's queries—it has no "outside data"—and so a naïve simulator might answer a random value, which we denote as $s_0$. In this case, if $\mathcal{D}$ queries $t_{-2}$ to the simulator, the simulator is stuck: it can see for itself that $\mathrm{RO}^2(t_{-2}) = t_0$ (it knows both $t_{-2}$ and $t_0$, at this point), so to be consistent it knows that it must answer a value $s_{-2}$ such that $\mathrm{RO}^2(s_{-2}) = s_0$ (indeed, the identity $\mathrm{RO} = g^2$ implies $\mathrm{RO}^2(g) = g(\mathrm{RO}^2)$), but finding such a value $s_{-2}$ is computationally infeasible for the simulator, given that $s_0$ is random.

Considering that the above attack can be generalized by having $\mathcal{D}$ build a long chain $t_{-\ell}, \ldots, t_0$ such that $t_{i+1} = \mathrm{RO}(t_i)$ before querying $\mathcal{S}$ at $t_0$ and then at $t_{-\ell}$, we conclude that the simulator should prepare a chain of RO queries of length[4] $q_1$, the number of RO-queries available to $\mathcal{D}$. More precisely, when the simulator receives its first query $t_0$ it chooses a random value $s_{-q_1}$, computes $s_{i+1} = \mathrm{RO}^i(s_i)$ for $-q_1 \leq i < 0$, and sets $g(t_0) = s_0$. Later, if $\mathcal{D}$ queries the simulator at any point $t_{-i}$, $0 \leq i \leq q_1$, the simulator can notice that $\mathrm{RO}^i(t_{-i}) = t_0$ (by iterating $\mathrm{RO}(\cdot)$ on $t_{-i}$ until it sees $t_0$), and then answer $s_{-i}$. (Note in passing that while the adversary $\mathcal{D}$ only has $q_1$ RO-queries to make, the simulator does not know for which points the adversary has prepared a long chain. Hence by querying the simulator at $\approx q_2$ random points among which is inserted the endpoint $t_0$ of some RO-chain of length $\approx q_1$, the adversary can force the simulator to make $\approx q_1q_2$ queries, to prepare a chain for each point. This is the intuition behind Theorem 3.2.)

[4]One could argue that a chain of length only $q_1/2$ is necessary, given that the adversary must also check the parallel chain to complete its attack; however, such fiddling is not worth the cost of complications in the proof.

Note that once the simulator determines $g(t_0) = s_0$ this also determines, for example, that $g(s_0) = g^2(t_0) = \mathrm{RO}(t_0)$; thus the value $t_1 := g(s_0)$ is fixed; likewise the value $s_1 := g(t_1)$ is fixed because $s_1 = g^2(s_0) = \mathrm{RO}(s_0)$, and so on. Thus if the adversary subsequently computes, for example, $t_{10} := \mathrm{RO}^{10}(t_0)$ on its own and then queries $t_{10}$ to the simulator, the simulator should notice that $t_{10} = \mathrm{RO}^{10}(t_0)$ and answer $\mathrm{RO}^{10}(s_0)$. The simulator can simplify its life in this regard if, right after setting $g(t_0) = s_0$, it precomputes the values $t_1, \ldots, t_{q_1+1}$ and $s_1, \ldots, s_{q_1}$ where $t_i := \mathrm{RO}^i(t_0)$, $s_i := \mathrm{RO}^i(s_0)$, setting $g(t_i) = s_i$ and $g(s_i) = t_{i+1}$ in the process (see Figure 6). Then, subsequently, it will "automatically" know that $g(t_{10}) = s_{10}$ without having to "notice" that $t_{10} = \mathrm{RO}^{10}(t_0)$.

We call the sequence of values $t_0, s_0, t_1, \ldots, s_{q_1}, t_{q_1+1}$ a *g-chain*, since $g(t_i) = s_i$ and $g(s_i) = t_{i+1}$. We note such a chain is just long enough that $\mathcal{D}$ cannot compute the endpoint $t_{q_1+1}$ "on its own" (using only RO queries), having only queried the simulator at $s_0$, and at no further points along the chain[5]. Moreover, each time $\mathcal{D}$ queries $\mathcal{S}$ at a point in such a chain, $\mathcal{S}$ extends the chain to always keep the furthest point of the chain at least $(q_1 + 1)$ RO queries away from the last input queried by $\mathcal{D}$, thus keeping the $g$-chain's endpoint permanently unknown to $\mathcal{D}$.

Altogether, three possible scenarios can play out when our simulator $\mathcal{S}$ answers a query $x$: (a) if $\mathcal{S}$ finds that it has already chosen a value for $g(x)$, it returns this value, and possibly extends the $g$-chain containing the query; (b) otherwise, (for this case picture $x$ as $t_{-i}$ above, for some $1 \leq i \leq \ell$), if $\mathcal{S}$ finds that $g(\mathrm{RO}^i(x))$ is defined for some $1 \leq i \leq q_1$, it answers $y := \mathrm{RO}^{-i}(g(\mathrm{RO}^i(x)))$ if it can compute this value from the previous queries it has made to RO, and aborts otherwise if it cannot; moreover, if it does not abort, then using its newly defined input-output pair $g(x) = y$ it precomputes a $g$-chain of length $2q_1 + 1$ starting at $x$ (this chain will actually "rear-end" a pre-existing $g$-chain, with no damage, since $g(\mathrm{RO}^i(x))$ is already defined); (c) if neither (a) nor (b) occur, the simulator computes a chain $s_{-q_1}, \ldots, s_{-1}, s_0, \ldots, s_{q_1}$ where $s_{-q_1}$ is chosen randomly and where $s_{-q_1+i} = \mathrm{RO}^i(s_{-q_1})$ for $i \geq 1$, then computes[6] the chain $t_0, \ldots, t_{q_1+1}$ where $t_i = \mathrm{RO}^i(x)$, and finally sets[7] $g(t_i) = s_i$, $g(s_i) = t_{i+1}$ for $0 \leq i \leq q_1$, before returning $s_0$ $(= g(t_0))$.

We call the query structure resulting from scenario (c) above a *ladder*; see Figure 6 (A). When the adversary queries a point $t_{-i}$ to the simulator, where $\mathrm{RO}^i(t_{-i}) = t_0$, the simulator "fills in" the values $g(t_{-j}) = s_{-j}$ and $g(s_{-j}) = t_{-j+1}$ for $-i \leq -j < 0$ (Figure 6 (B)), corresponding to scenario (b) above. Note that if the adversary queries a point $s_{-j}$ to the simulator before querying any point $t_{-k}$ with $k \geq j$, the simulator will abort, being unable to invert RO beyond the leftmost point $t_{-i}$ such that the adversary has queried $t_{-i}$ to the simulator (Figure 6 (C)). However, this latter event happens with low probability given that the adversary must guess the value $s_{-j}$ out of thin air: indeed, the adversary's only way of discovering $s_{-j}$ is to query $\mathcal{S}$ at $t_{-i}$ for some $i \geq j$ (this is formally argued in the proof). Finally, a case not depicted in Figure 6 occurs when the adversary queries a point $t_i$ with $i > 0$ or $s_i$ with $i \geq 0$. This corresponds to scenario (a): the simulator will return the predetermined value of $g$, after "extending" the ladder such that the ladder's furthest point is at a distance of at least $(q_1 + 1)$ RO calls from the last point queried by $\mathcal{D}$. A full specification of our simulator is given in Figure 5.

The above outlines the simulator $\mathcal{S}$. As for the indifferentiability proof itself, we use a sequence of games, where each game presents the adversary with a two-oracle environment. In the first game, the environment is equivalent to the pair $(\mathrm{RO}, \mathcal{S})$; in the last, to the pair $(G[g], g)$ for a random $g$. For each pair of adjacent games $\mathrm{G}_i$, $\mathrm{G}_{i+1}$, the adversary $\mathcal{D}$'s distinguishing advantage $\Pr[\mathcal{D}^{\mathrm{G}_i} \Rightarrow 1] - \Pr[\mathcal{D}^{\mathrm{G}_{i+1}} \Rightarrow 1]$ is upper bounded, where $\mathcal{D}^{\mathrm{G}_i}$ notates $\mathcal{D}$ run in the two-oracle environment of game $\mathrm{G}_i$. Summing these

[5] This does not preclude $\mathcal{D}$ from knowing beforehand the value of some $s_i$'s with $i > 0$, since it could make these RO queries before querying the simulator at $s_0$, but this costs $\mathcal{D}$ the same number of RO queries as computing these values afterward.

[6] In fact, the values $t_0, \ldots, t_{q_1+1}$ are already computed during step (b), given that step (b) is unsuccessful.

[7] Note one could also set $g(s_{-1}) = t_0$, but this changes little.

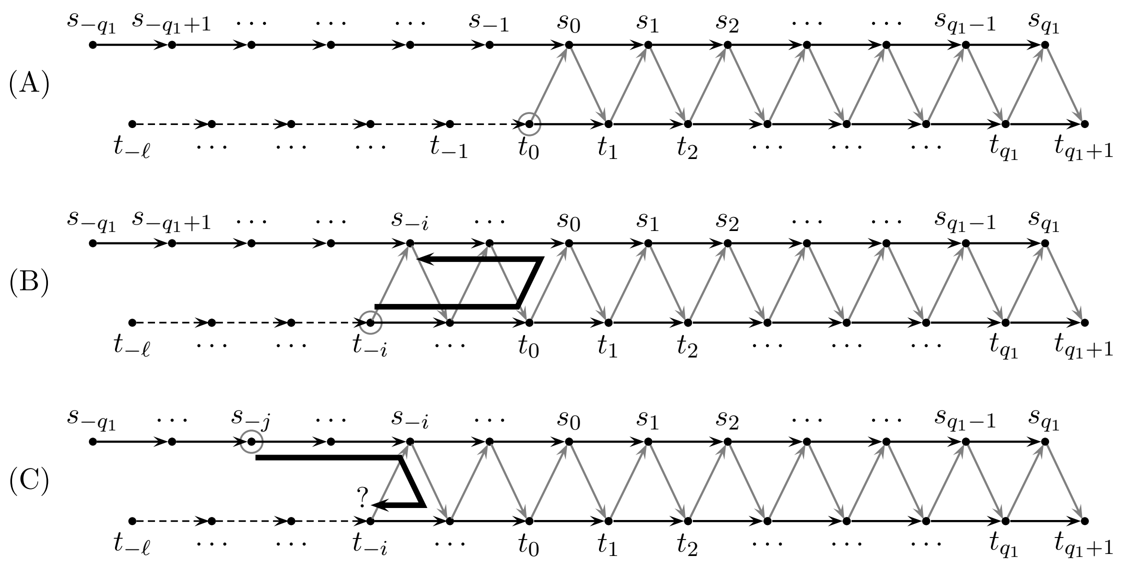


Figure 6: Illustration of the simulator for Theorem 3.3. Horizontal arrows show queries to RO, while upward and downard diagonal arrows show $g$-queries defined by the simulator. Dashed arrows show RO queries known only to the attacker. The last point queried by the attacker and answered by the simulator is circled. Bold black arrows indicate the internal path followed by the simulator to answer the attacker's last query. Top (A): The simulator builds a ladder after being queried at $t_0$. Middle (B): The simulator "fills in" values of $g(\cdot)$ after being queried at $t_{-i}$. Bottom (C): The simulator aborts after being queried at $s_{-j}$.

upper bounds gives the final indifferentiability bound. Our proof uses 23 games in all.

## 4 HMAC as a General-purpose Keyed Hash Function

HMAC [5] uses a hash function to build a keyed hash function, i.e. one that takes both a key and message as input. Fix some hash function[8] $H\colon \{0,1\}^* \to \{0,1\}^n$. HMAC assumes this function $H$ is built by iterating an underlying compression function with a message block size of $d \geq n$ bits. We define the following functions:

$$\begin{aligned} F_K(M) &= H((\rho(K) \oplus \mathsf{ipad}) \,\|\, M) \\ G_K(M) &= H((\rho(K) \oplus \mathsf{opad}) \,\|\, M) \end{aligned} \qquad \text{where } \rho(K) = \begin{cases} H(K) & \text{if } |K| > d \\ K & \text{otherwise.} \end{cases}$$

The two constants used are $\mathsf{ipad} = \mathsf{0x36}^{d/8}$ and $\mathsf{opad} = \mathsf{0x5c}^{d/8}$. These constants are given in hexadecimal, translating to binary gives $\mathsf{0x36} = \mathsf{0011\,0110}_2$ and $\mathsf{0x5c} = \mathsf{0101\,1100}_2$. Recall that we have defined the $\oplus$ operator so that, if $|K| < d$, it first silently pads out the shorter string by sufficiently many zeros before computing the bitwise xor. It will also be convenient to define $\mathsf{xpad} = \mathsf{ipad} \oplus \mathsf{opad}$. The function $\mathrm{HMAC}\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ is defined by

$$\mathrm{HMAC}(K, M) = G_K(F_K(M)) = (G_K \circ F_K)(M)\ .$$

We sometimes write $\mathrm{HMAC}_d[P]$, $\mathrm{HMAC}_d$, or $\mathrm{HMAC}[P]$ instead of HMAC when we want to make the reliance on the block size and/or an underlying ideal primitive explicit.

Some applications of HMAC. The HMAC construction was originally suggested for use as a PRF or MAC — settings in which $K$ is a secret value generated by a trusted party. And while HMAC is still used as such, it has also come to be used (or suggested for use) in settings where $K$ is public [25, 29, 32] or a non-uniform secret [36]. There also exist settings whose theoretical security models allow adversarially generated keys [38].

- *HKDF.* The HKDF scheme builds from HMAC a general key derivation function [32]. It is currently in the process of being standardized [31]. The construction follows an extract-then-expand approach. In the extract phase, compute $Y = \mathrm{HMAC}(sa, msk)$ using a public, application-dependent salt $sa$ and a secret source of entropy $msk$. It is strongly recommended [31] that $sa$ be a uniformly selected $n$-bit string. In the expand phase, a key $K = \mathrm{HMAC}(Y, da)$ is derived using application-specific context information $da$. IPSec key exchange similarly utilizes HMAC with public information as key [25, 29].
- *TLS.* The TLS protocol uses HMAC as a key-derivation function. One derives a key as $\mathrm{HMAC}(pms, da)$ where $da$ consists of public nonces and public session-dependent information and $pms$ is either a 48 byte string consisting of a version number and 46 random bytes (when using RSA key transport) or a Diffie-Hellman value (when using Diffie-Hellman key exchange) of length specified by the group size. The value $pms$ is therefore secret but not necessarily a secret, uniform bit string from an adversary's perspective.
- *PKCS#5.* The PKCS#5 standard uses HMAC for password-based key derivation. A derived key for some password $pw$ and salt $sa$ is defined by $K = \bigoplus_{i=1}^{\ell} \mathrm{HMAC}^{\ell}(pw, sa)$. In practice, passwords are chosen by users in all kinds of ways, and typically have relatively little entropy.

[8]RFC 2104 defines HMAC over strings of bytes, but we chose to use bits to provide more general positive results — all our negative results lift to a setting in which only byte strings are used. Note also that for simplicity we assumed $H$ with domain $\{0,1\}^*$. In practice hash functions often do have some maximal length (e.g., $2^{64}$), and in this case HMAC must be restricted to smaller lengths.

- *Hedged cryptography.* Ristenpart and Yilek [38] suggest a general way of using HMAC to modify cryptographic routines so as to hedge against randomness failures. For example, let $\mathcal{E}(pk, M\,;\,R)$ denote encrypting message $M$ under public-key $pk$ and using randomness $R$. They suggest to modify encryption to instead proceed via $\mathcal{E}(pk, M\,;\,\mathrm{HMAC}(R, pk \,\|\, M))$. (Here we show only the case that $|R| = n$.) This merges two prior suggestions due to Bellare et al. [4] and Yilek [42]. One of the security goals targeted is chosen-distribution attack (CDA) security [4], which requires no partial information about high min-entropy messages is leaked even in the presence of adversarially-specified randomness $R$.

Common to the examples above is that security proofs of HMAC when used with uniform secret keys [3, 5] do not (directly) apply. In some cases, standard model proofs have nevertheless been given. Particularly, positive results about HMAC's security as a randomness extractor [15, 17, 23, 32], which are applicable to its use in HKDF, IPsec, and TLS. But in other cases it seems unlikely that standard model analyses are possible. The security of TLS key transport is one example, as discussed in [35]. Another example is when using lower entropy secrets with HKDF, as discussed in [32]. The same situation faces PKCS#5 because passwords are often short. In these two settings, the results of [15, 17, 23, 32] are inapplicable because they require sources with high (computational) min-entropy. Finally, security proofs for the hedged encryption construction seems to fundamentally rely on the "programmability" of ROs [4, 38].

Prior work has therefore turned to assuming HMAC is a keyed RO to achieve positive results. A variant of TLS key exchange is analyzed under this assumption by Morissey et al. [35]. Krawczyk analyzes HKDF as a randomness extractor under this assumption [32]. Ristenpart and Yilek analyze the security of the hedged encryption construction under this assumption [38]. Validating this assumption as used in these works would require proving indifferentiability of HMAC from a keyed RO for distinguisher's that query arbitrary keys.

USING INDIFFERENTIABILITY. Our focus is these last analyses in the keyed ROM. We want to understand if the positive results carry over to a setting in which the structure of HMAC is taken into account. Namely, when one uses HMAC$[P]$ for some underlying ideal primitive $P$ (e.g., an ideal compression function). This is important because security proofs taking into account HMAC's structure would rule out subtle, exploitable interactions between HMAC's design and its use in these applications.

Indifferentiability provides a mechanism for this kind of analysis: should HMAC$[P]$ be indifferentiable from a keyed RO $\mathcal{R}$ with good bounds, then the MRH composition theorem [33] could be applied to theorems from [32, 35, 38] to give corollaries that security holds when using HMAC$[P]$.[9] To simultaneously cover all the settings of [32, 35, 38], indifferentiability would have to hold even for adversaries that can query arbitrary keys and messages.

In the following sections, we will therefore analyze the security of HMAC in the sense of being indifferentiable from a keyed RO. As we will see, the story is more involved than one might expect.

## 4.1 Weak Key Pairs in HMAC

Towards understanding the indifferentiability of HMAC, we start by observing that the way HMAC handles keys gives rise to two worrisome classes of weak key pairs.

- **Colliding key pairs:** We say that keys $K \neq K'$ *collide* if
$$\rho(K) \,\|\, 0^{d-|\rho(K)|} \;=\; \rho(K') \,\|\, 0^{d-|\rho(K')|}\;.$$

[9] Technically, this would only apply to a slightly weaker security model for [38] that does not allow hash-dependent randomness, message distributions. See [37] for a discussion.

For any message $M$ and colliding keys $K, K'$ it holds that $\mathrm{HMAC}(K, M) = \mathrm{HMAC}(K', M)$.

Colliding keys exist because of HMAC's ambiguous encoding of different-length keys. Examples of colliding keys include any $K, K'$ for which $|K| < d$ and $K' = K \parallel 0^s$ where $1 \leq s \leq d - |K|$. Or any $K, K'$ such that $|K| > d$ and $K' = H(K)$. As long as $H$ is collision-resistant, two keys of the same length can never collide.

Colliding keys enable a simple attack against indifferentiability: query $\mathsf{Func}$ on $(K, M)$ and $(K', M)$ for $K, K'$ colliding and see if the outputs are equal. Colliding key pairs may also have implications for other settings. We discuss this all more in the next section.

The second form of weak key pair we term ambiguous:

- **Ambiguous key pairs:** A pair of keys $K \neq K'$ is *ambiguous* if
$$\rho(K) \oplus \mathsf{ipad} \;=\; \rho(K') \oplus \mathsf{opad}\ .$$
For any $X$, both $F_K(X) = G_{K'}(X)$ and $G_K(X) = F_{K'}(X)$ when $K, K'$ are ambiguous.

An example such pair is $K, K'$ of length $d$ bits for which $K \oplus K' = \mathsf{xpad}$. For any key $K$, there exists one key $K'$ that is easily computable and for which $K, K'$ are ambiguous: set $K' = \rho(K) \oplus \mathsf{xpad}$. Finding a third key $K''$ that is also ambiguous with $K$ is intractable should $H$ be collision resistant. The easily-computable $K'$ will not necessarily have the same length as $K$. In fact, there exist ambiguous key pairs of the same length $k$ only when $k \in \{d-1, d\}$. For a fixed length shorter than $d-1$, no ambiguous key pairs exist due to the fact that the second least significant bit of $\mathsf{xpad}$ is 1. For a fixed length longer than $d$ bits, if $n < d-1$ then no ambiguous key pairs exist and if $n \geq d-1$ then producing ambiguous key pairs would require finding $K, K'$ such that $H(K) \oplus H(K')$ equals the first $n$ bits of $\mathsf{xpad}$. This is intractable for any reasonable hash function $H$.

Unlike colliding key pairs, ambiguous key pairs, at first glance, may not seem to be problematic for security. But in fact they give rise to a chain-shift-like property for HMAC that, as with $H^2$, can lead to insecurities in some settings. We explore this in Section 4.3.

Summary. We uncover two types of weak key pairs in HMAC. We will discuss further in Section 4.1 how colliding keys trivially rule out indifferentiability. Colliding key pairs are avoided by using fixed-length keys. However, even here, we have ambiguous key pairs. We will show in Section 4.3 how HMAC when ambiguous key pairs are allowed cannot be proven indifferentiability with good concrete security. The underlying structural issue is similar to that of second iterate constructions. We leave as an open question showing a weak upper bound for the indifferentiability of HMAC with ambiguous key pairs, but suspect that the techniques from the proof of Theorem 3.3 might be applicable.

Finally, and fortuitously, most applications of HMAC appear to avoid both kinds of weak key pairs. As we will show in Section 4.4, we can prove indifferentiability holds with standard, good bounds for some cases in which weak key pairs are avoided. For example, the common case of using keys of a fixed length less than $d-1$ provides security. A summary of all these results is given in Figure 7.

## 4.2 Colliding Key Pairs and the Indifferentiability of HMAC

Colliding key pairs give rise to a simple attack against the indifferentiability of HMAC. We have the following theorem:

**Theorem 4.1** Let $\mathrm{HMAC}[H[P]]$ be the HMAC construction for an arbitrary underlying hash function $H[P]$ and let $\mathcal{R}$ be a keyed RO. Then there exists an adversary $\mathcal{A}$ making two queries and running in a small constant amount of time such that for any simulator $\mathcal{S}$ it holds that
$$\mathbf{Adv}^{\mathrm{indiff}}_{\mathrm{HMAC}[H[P]],\mathcal{R},\mathcal{S}}(\mathcal{A}) \;\geq\; 1 - \frac{1}{2^n}\ .\ \square$$

| Key space includes | Indifferentiable | Queries | Section |
|---|---|---|---|
| Colliding key pairs | No | 2 | §4.1 |
| Ambiguous key pairs | At most weakly | $\mathcal{O}(2^{n/4})$ | §4.3 |
| Only keys $K$ of fixed length $\|K\| < d-1$ | Yes | $\mathcal{O}(2^{n/2})$ | §4.4 |

Figure 7: Summary of indifferentiability of HMAC from a keyed RO for various restrictions on the key space. The "Queries" column indicates the number of queries used by an attacker to gain good advantage against any simulator.

**Proof:** Distinguisher $\mathcal{A}$ first picks two keys $K \neq K'$ that collide and picks an arbitrary message $M$. It then queries its $\mathsf{Func}$ oracle on $(K, M)$ and $(K', M)$ to retrieve two values $Y, Y'$. If $Y = Y'$ then it returns 1 (guessing that it is in game $\mathrm{Real}_{\mathrm{HMAC}[H[P]],\mathcal{R}}$) and returns 0 otherwise (guessing that it is in game $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}$). The advantage of $\mathcal{A}$ is equal to $1 - 2^{-n}$ regardless of the simulator $\mathcal{S}$, which is never invoked. ∎

Colliding key pairs endanger security of any application of HMAC that uses variable-length keys. The formal security of several applications is ruled out when colliding key pairs are allowed. For PKCS#5, consider the slight simplification of using HMAC($pw, sa$) to derive a key from a password $pw$ and salt $sa$. Then if the set of passwords from which $pw$ was drawn includes colliding key pairs (which is allowed by, e.g., the password-based key derivation function security definition given by [9]), then the search space of a dictionary attack will be reduced. For hedged cryptography, if one extends the security notion of [4] to allow the adversary to pick variable lengths of randomness, then there exists a simple adversary that violates the chosen-distribution attack security of the HMAC-based constructions from [38].

That said, we are unaware of any exploitable security vulnerabilities in practice due to colliding key pairs. The PKCS#5 example above would require having passwords in the set that are encoded to binary strings that end in zero bytes. HMAC as actually used in HKDF, TLS, and hedged cryptography all use fixed-length keys.

RELATED-KEY ATTACKS. We digress for a moment to consider the setting of related-key attacks (RKAs) against HMAC as a PRF (i.e., using uniformly selected, secret keys). Recall that RKA-PRF security [6] asks that no attacker can distinguish between two oracles to which it can make adaptive queries. The first oracle allows the attacker to query a related key function $\phi$ from some allowed set $\Phi$ and a message $M$. It returns HMAC($\phi(K), M$) for a randomly chosen key $K$. The second oracle has the same interface but returns $\rho(\phi(K), M)$ where $\rho$ is a family of random functions.

Colliding keys give rise to an RKA against HMAC for any $\Phi$ that includes both the identity function and a function $\phi$ for which the keys $K, \phi(K)$ collide. The adversary queries the identity function and a message $M$ and in a second query $\phi$ and the same message $M$. If the returned values are the same it guesses that it is interacting with HMAC and otherwise it guesses it is interacting with $\rho$. The adversary achieves advantage $1 - 2^{-n}$.

### 4.3 Ambiguous Key Pairs and the Indifferentiability of HMAC

We now turn to ambiguous key pairs and show that these also lead to lower bounds on the indifferentiability of HMAC. Recall that with $H^2$, problems arose because outputs of $H^2$ on some message were valid intermediate values used in computing $H^2$ on some other message. HMAC is the same due to ambiguous key pairs. Let $M$ be some message and $K, K'$ be an ambiguous key pair. Then, we have

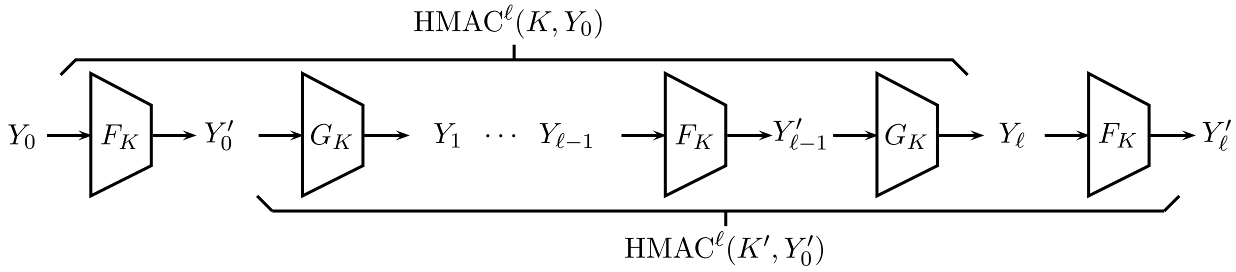


Figure 8: Diagram of two hash chains $(K, Y) = (Y_0, \ldots, Y_\ell)$ and $(K', Y') = (Y'_0, \ldots, Y'_\ell)$ for HMAC where $\rho(K') = \rho(K) \oplus \mathsf{xpad}$.

that $\rho(K') = \rho(K) \oplus \mathsf{xpad}$ and so $F_K(M) = G_{K'}(M)$. Thus,

$$\mathrm{HMAC}(K', F_K(M)) = G_{K'}(\mathrm{HMAC}(K, M))$$

This property does not appear immediately exploitable in attacks against, for example, the HMAC applications mentioned above. We will thus follow the same path as we did with $H^2$ to highlight how this structural property affects the ability to show that HMAC is indifferentiable from a RO — even when colliding key pairs do not arise. The result will be our ruling out strong indifferentiability bounds when ambiguous key pairs arise, and thus limiting the scope of applicability of composition-based proofs of security for applications of HMAC. As with $H^2$, we will also detail HMAC applications in which ambiguous key pairs can be exploited by attackers. As before, these revolve around hash chains.

Keyed hash chains. We lift our notions of hash chains from Section 3 to the setting of keyed hash functions. Let $H\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ be a keyed hash function. A hash chain $Y = (K, Y_0, \ldots, Y_\ell)$ is a key $K$, a message $Y_0$, and a sequence of $\ell$ values $Y_i = H(K, Y_{i-1})$ for $1 \le i \le \ell$. So a keyed hash chain $Y = (K, Y_0, \ldots, Y_\ell)$ for HMAC has $Y_i = \mathrm{HMAC}(K, Y_{i-1})$ for $1 \le i \le \ell$. We refer to $Y_0$ as the *start* of the keyed hash chain and to $Y_\ell$ as the *end*. Two keyed hash chains $Y, Y'$ are *non-overlapping* if $Y_i \ne Y'_j$ for all $0 \le i \le j \le \ell$.

Let $\mathrm{HMAC}[P](K, M)$ be HMAC using as underlying hash function a random oracle $P\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ and extend the definitions of $\rho, F_K, G_K$ to use $P$ in the natural way. Given the start and end of a chain $Y = (K, Y_0, \ldots, Y_\ell)$, it is easy for an adversary to compute the start and end of a new chain $Y' = (K', Y'_0, \ldots, Y'_\ell)$. To do so, the adversary chooses $K'$ so that $\rho(K') = \rho(K) \oplus \mathsf{xpad}$ and then computes $Y'_0 \leftarrow F_K(M)$ and $Y'_\ell \leftarrow F_K(Y_0)$. By the choice of $K'$ it holds that both $F_K(X) = G_{K'}(X)$ and $G_K(X) = F_{K'}(X)$ for all $X$. Thus $Y_0, Y'_\ell$ are valid start and end points for a chain because

$$\begin{aligned}
(G_{K'} \circ F_{K'})^{\ell}(Y'_0) &= G_{K'}(F_{K'}(\cdots F_{K'}(Y'_0) \cdots)) \\
&= G_{K'}(F_{K'}(\cdots F_{K'}(F_K(M)) \cdots)) \\
&= F_K(G_K(\cdots G_K(F_K(M)) \cdots)) \\
&= F_K\big((G_K \circ F_K)^{\ell}(K, M)\big) \\
&= F_K(Y_\ell) \\
&= Y'_\ell
\end{aligned}$$

We refer to the equivalence above as the chain-shift property of HMAC. A diagram of the two HMAC chains involved appears in Figure 8. Finally, we note that with overwhelming probability (over the coins of $P$) $Y$ and $Y'$ will not overlap.

To capture the gap between HMAC and a RO in a formal way, we extend the $\mathrm{CHAIN}_{H[P],n,\ell}$ game

| | |
|---|---|
| **main** $\mathrm{CHAIN}_{H[P],k,n,\ell}$: | **procedure** $\mathsf{Prim}(M)$: |
| $i \leftarrow 0$ ; $K \leftarrow\!\!\$\ \{0,1\}^k$ | $i \leftarrow i+1$ |
| $Y_0 \leftarrow\!\!\$\ \{0,1\}^n$ ; For $i = 1$ to $\ell$ do $Y_i \leftarrow H[P](K, Y_{i-1})$ | If $i \geq \ell \cdot \mathsf{Cost}(H,n)$ then Ret $\perp$ |
| $(K', Y_0^*, Y_\ell^*) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(K, Y_0, Y_\ell)$ | Ret $P(M)$ |
| $Y_0' \leftarrow Y_0^*$ ; For $i = 1$ to $\ell$ do $Y_i' \leftarrow H[P](K', Y_{i-1}')$ | |
| $\mathcal{Y} \leftarrow \{Y_0, \ldots, Y_\ell\}$ | |
| $\mathcal{Y}' \leftarrow \{Y_0', \ldots, Y_\ell'\}$ | |
| If $(Y_\ell' = Y_\ell^*) \wedge ((\mathcal{Y} \cap \mathcal{Y}') = \emptyset)$ then Ret $\mathsf{true}$ | |
| Ret $\mathsf{false}$ | |

Figure 9: A keyed version of the chain-making game.

from Section 3 to work for keyed chains; see Figure 9. The only change is that a randomly chosen challenge key of $k$ bits is generated by the game and the adversary now outputs not only an attempted start and end point of a chain, but a chosen key as well. We define advantage as

$$\mathbf{Adv}^{\mathrm{chain}}_{H[P],k,n,\ell}(\mathcal{A}) \;=\; \Pr\left[\, \mathrm{CHAIN}^{\mathcal{A}}_{H[P],k,n,\ell} \Rightarrow \mathsf{true} \,\right] .$$

We compare the CHAIN security of HMAC$[P]$ to the CHAIN security of a keyed RO $\mathcal{R}$. Technically, we look at the the achieved security of $\mathrm{CHAIN}_{\mathrm{HMAC}[P],k,n,\ell}$ to that of $\mathrm{CHAIN}_{H[\mathcal{R}],k,n,\ell}$ where $H[\mathcal{R}](K,M) = \mathcal{R}(K,M)$ implements a keyed RO. Security in the latter case follows in the same manner that CHAIN security was established for a (non-keyed) RO, as per Claim 3.1. Namely,

$$\mathbf{Adv}^{\mathrm{chain}}_{H[\mathcal{R}],k,n,\ell}(\mathcal{A}) \leq \frac{(3\ell)^2}{2^n}$$

for all adversaries $\mathcal{A}$. On the other hand, an adversary $\mathcal{B}$ exploiting the chain-shift property of HMAC achieves

$$\mathbf{Adv}^{\mathrm{chain}}_{\mathrm{HMAC}[P],k,n,\ell}(\mathcal{B}) \geq 1 - \frac{\ell^2}{2^n} .$$

The adversary $\mathcal{B}$ lets $K' = \rho(K) \oplus \mathsf{xpad}$ and picks $Y_0' = F_K(Y_0)$ and $Y_\ell' = F_K(Y_\ell)$. As with $H^2$, we see a gap between the CHAIN security achieved by the ideal object (a keyed RO) and HMAC.

Note that $\mathcal{B}$ may output a $K'$ that is of different length than $K$. But the attack extends to a setting requiring $|K| = |K'|$ for some, but not all, choices of the parameter $k$. In particular, the adversary $\mathcal{B}$ can always find a suitable $K'$ with $|K'| = k$ for $k \in \{d-1, d\}$.

Mutual proofs of work with keys. We similarly lift mutual proofs of work to the setting of keyed chains. Referring to Figure 10, we modify the protocol to allow both parties to choose a key and a message as challenge for the other party. Likewise, the security game $\mathrm{POW}_{H[P],n,\ell_1}$ is modified as shown in the right hand side of Figure 10 to give game $\mathrm{POW}_{H[P],k,n,\ell_1}$. An adversary can use the chain-shift-like property of HMAC to mount a successful attack against $\mathrm{POW}_{\mathrm{HMAC}[P],k,n,\ell_1}$ for any $\ell_1 > 1$ and for $P$ a RO. Consider the following adversary $\mathcal{A}$. When it first receives a nonce $K_2, X_2$, it chooses $K_1$ such that $\rho(K_1) = \rho(K_2) \oplus \mathsf{xpad}$ and lets $X_1 \leftarrow F_{K_2}(X_2)$. Later when it receives $Y_1 = \mathrm{HMAC}^{\ell_1}[P](K_1, X_1)$ it computes its response as $Y_2 \leftarrow F_{K_1}(Y_1)$, sets $\ell_2 = \ell_1 + 1$, and returns $\ell_2, Y_2$.

| $\mathcal{P}_1$ | | $\mathcal{P}_2$ |
|---|---|---|
| $K_2 \leftarrow\!\!\$\ \{0,1\}^k$ ; $X_2 \leftarrow\!\!\$\ \{0,1\}^n$ | $\xrightarrow{K_2, X_2}$ | $K_1 \leftarrow\!\!\$\ \{0,1\}^k$ ; $X_1 \leftarrow\!\!\$\ \{0,1\}^n$ |
| | $\xleftarrow{K_1, X_1}$ | |
| $Y_1 \leftarrow H^{\ell_1}(K_1, X_1)$ | $\xrightarrow{\ell_1, Y_1}$ | $Y_2 \leftarrow H^{\ell_2}(K_2, X_2)$ |
| | $\xleftarrow{\ell_2, Y_2}$ | |
| $\hat{\mathcal{Y}}_1 \leftarrow \{H^i(K_1, X_1) \mid 0 \le i \le \ell_1\}$ | | $\hat{\mathcal{Y}}_1 \leftarrow \{H^i(K_1, X_1) \mid 0 \le i \le \ell_1\}$ |
| $\hat{\mathcal{Y}}_2 \leftarrow \{H^i(K_2, X_2) \mid 0 \le i \le \ell_2\}$ | | $\hat{\mathcal{Y}}_2 \leftarrow \{H^i(K_2, X_2) \mid 0 \le i \le \ell_2\}$ |
| $Y_2' \leftarrow H^{\ell_2}(K_2, X_2)$ | | $Y_1' \leftarrow H^{\ell_1}(K_1, X_1)$ |
| Ret $(Y_2' = Y_2) \wedge (\hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$ | | Ret $(Y_1' = Y_1) \wedge (\hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$ |

**main** $\mathrm{POW}_{H[P],n,\ell_1}$:
$K_2 \leftarrow\!\!\$\ \{0,1\}^k$
$X_2 \leftarrow\!\!\$\ \{0,1\}^n$
$(K_1, X_1) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(K_2, X_2)$
$Y_1 \leftarrow H^{\ell_1}[P](K_1, X_1)$
$(\ell_2, Y_2) \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Prim}}(\ell_1, Y_1)$
$\hat{\mathcal{Y}}_1 \leftarrow \{H^i[P](K_1, X_1) \mid 0 \le i \le \ell_1\}$
$\hat{\mathcal{Y}}_2 \leftarrow \{H^i[P](K_2, X_2) \mid 0 \le i \le \ell_2\}$
$Y_2' \leftarrow H^{\ell_2}[P](K_2, X_2)$
If $q \ge \ell_2 \cdot \mathsf{Cost}(H, n)$ then
  Ret $\mathsf{false}$
Ret $(Y_2' = Y_2 \wedge \hat{\mathcal{Y}}_1 \cap \hat{\mathcal{Y}}_2 = \emptyset)$

**subroutine** $\mathsf{Prim}(u)$
$q \leftarrow q + 1$ ; Ret $P(u)$

Figure 10: Example protocol **(left)** and adversarial $\mathcal{P}_2$ security game **(right)** for mutual proofs of work with keyed hash functions.

By the chain-shift property of HMAC, we get that

$$\begin{aligned} Y_2 &= F_{K_1}\big((G_{K_1} \circ F_{K_1})^{\ell_1}(X_1)\big) \\ &= F_{K_1}\big((G_{K_1} \circ F_{K_1})^{\ell_1}(F_{K_2}(X_2))\big) \\ &= F_{K_1}(G_{K_1}(\cdots F_{K_1}(F_{K_2}(X_2))\cdots)) \\ &= G_{K_2}(F_{K_2}(\cdots G_{K_2}(F_{K_2}(X_2))\cdots)) \\ &= \mathrm{HMAC}^{\ell_1+1}[P](K_2, X_2)\ . \end{aligned}$$

The two chains will be non-overlapping with probability close to one and $\mathcal{A}$ makes at most two $P$-applications, in turn ensuring that $q < 2\ell_2$ when $\ell_1 > 1$.

Direct indifferentiability lower bounds. All the above suggests that HMAC is not indifferentiable from a keyed RO because of ambiguous key pairs and regardless of the strength of the primitive underlying HMAC. As before, this requires careful interpretation, in the same sense as discussed for $H^2$ in Section 3. We therefore provide a direct indifferentiability adversary. It relies on the quantity $p'(H, w, \ell)$ that represents the probability of certain collisions associated to the queries the distinguisher makes. The quantity depends on the particulars of the construction $H[P]$ and is defined as

$$p'(H, w, \ell) \;=\; \Pr\left[\,\{H(\mathsf{ipad} \,\|\, U_1), \ldots, H(\mathsf{ipad} \,\|\, U_{w-1})\} \cap \{y_0, y_1, \ldots, y_\ell\} \neq \emptyset\,\right]\ ,$$

for $U_1, \ldots, U_{w-1}$ being $w-1$ independent $n$-bit strings, and $y_0, y_1, \ldots, y_\ell$ are out of a chain of $n$-bit values $x_0, y_0, \ldots, x_\ell, y_\ell$ where $x_0$ is chosen at random, and $y_i = H(\mathsf{ipad} \,\|\, x_i)$ for $i = 0, \ldots, \ell$ and $x_i = H(\mathsf{opad} \,\|\, y_{i-1})$ for $i = 1, \ldots, \ell$. We show in Appendix A that, for typical $H$ constructions, $p'(H, w, \ell) \le 2(w-1)(\ell+2)/2^n + 2(\ell+1)^2/2^n$.

**Theorem 4.2 [Attack against** HMAC] Let $H[P]$ be an arbitrary hash construction with $n$-bit outputs, calling a random primitive $P$, and let $\mathcal{R} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ be a keyed random oracle. For all integer parameters $w, \ell \ge 1$, there exists an adversary $\widetilde{\mathcal{D}}_{w,\ell}$ making $2\ell$ $\mathsf{Func}$-queries and

$(w+1) \cdot \mathsf{Cost}(H, n+d)$ $\mathsf{Prim}$-queries such that for all simulators $\mathcal{S}$,

$$\mathbf{Adv}^{\mathrm{indiff}}_{\mathrm{HMAC}[H[P]],\mathcal{R},\mathcal{S}}(\widetilde{\mathcal{D}}_{w,\ell}) \geq 1 - p'(H, w, \ell) - \frac{5\ell^2}{2^{n+1}} - \frac{q_{\mathcal{S}}\ell}{2^n} - \frac{q_{\mathcal{S}}}{2^n} - \frac{q_{\mathcal{S}}}{w \cdot \ell} - \frac{1}{w} \,,$$

where $q_{\mathcal{S}}$ is the overall number of $\mathcal{R}$ queries by $\mathcal{S}$ when replying to $\widetilde{\mathcal{D}}_{w,\ell}$'s $\mathsf{Prim}$ queries. ▌

The proof is given in Appendix C. The interpretation of the theorem is analogous to that of Theorem 3.2.

### 4.4 Indifferentiability Upper Bound for HMAC with Restricted Keys

We have seen that HMAC's construction gives rise to two kinds of weak key pairs that can be abused to show that HMAC is not indifferentiable from a keyed RO (with good bounds). But weak key pairs are serendipitously avoided in most applications. For example, the recommended usage of HKDF [32] specifies keys of a fixed length less than $d-1$. Neither kind of weak key pairs exist within this subset of the key space. For the other applications mentioned at the beginning of Section 4, our positive results should be applicable, with some caveats: PKCS#5 for passwords sufficiently restricted, hedged cryptography with certain lengths of randomness, and TLS for particular premaster secret sizes.

Our positive results will focus primarily on the case mentioned above. That is, we restrict attention to keys $K$ for which $|K| = k$ and $k$ is a fixed integer less than $d-1$. In fact, we prove a positive indifferentiability bound for a slightly more general key space, described next.

Restricted key spaces for HMAC. We first provide some definitions regarding restricted key spaces for HMAC. We here focus on the case where all keys are of length $d$ or less. Let $\mathcal{K} \subseteq \{0,1\}^{\leq d}$ be a set of keys. We say $\mathcal{K}$ is *allowed* if there exists a function $\mathsf{GetKey}\colon \{0,1\}^d \to \{0,1\}^*$ such that

$$\mathsf{GetKey}(K \oplus \mathsf{ipad}) \;=\; K \qquad \text{and} \qquad \mathsf{GetKey}(K \oplus \mathsf{opad}) \;=\; K \,.$$

The function $\mathsf{GetKey}$ implies the existence of a predicate $\mathsf{IsOuter}\colon \{0,1\}^d \to \{0,1\}$ such that for any $K \in \mathcal{K}$,

$$\mathsf{IsOuter}(K \oplus \mathsf{ipad}) \;=\; 0 \qquad \text{and} \qquad \mathsf{IsOuter}(K \oplus \mathsf{opad}) \;=\; 1 \,.$$

The predicate $\mathsf{IsOuter}$ can determine which pad was used with a key, while the function $\mathsf{GetKey}$ can invert xor'ing by $\mathsf{ipad}$ or $\mathsf{opad}$. For any allowed $\mathcal{K}$ it must be that $K \oplus \mathsf{ipad} \neq K' \oplus \mathsf{opad}$ for all $K, K' \in \mathcal{K}$. We let $\mathcal{K}$-restricted HMAC be the function $\mathrm{HMAC}\colon \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$ that is undefined for $K \notin \mathcal{K}$ and for $K \in \mathcal{K}$ is defined equivalently to $\mathrm{HMAC}\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$. In our proofs below, we assume that indifferentiability adversaries do not query keys $K \notin \mathcal{K}$ to the $\mathsf{Func}$ oracle.

One example of an allowed $\mathcal{K}$ is all keys $K$ of length equal to $d$ with second least significant bit equal to 0. For this class, $\mathsf{IsOuter}(X) = 1$ iff the second least significant bit of $X$ is 0. (Recall that the second least significant bit of $\mathsf{ipad}$ is 1 and of $\mathsf{opad}$ is 0.) Another example is exactly the set of widest consequence: the set of all keys of a fixed length that is less than $d-1$.

HMAC using a RO. We start with the simpler case, proving that $\mathcal{K}$-restricted HMAC is indifferentiable from a keyed RO when the underlying hash function is modeled as a RO.

**Theorem 4.3** Fix $d, n > 0$. Let $P\colon \{0,1\}^* \to \{0,1\}^n$ be a RO, and consider $\mathcal{K}$-restricted $\mathrm{HMAC}_d[P]$ for an allowed key set $\mathcal{K}$. Let $\mathcal{R}\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ be a keyed RO. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{A}$ whose total query cost is $\sigma$ it holds that

$$\mathbf{Adv}^{\mathrm{indiff}}_{\mathrm{HMAC}_d[P],\mathcal{R},\mathcal{S}}(\mathcal{A}) \leq \mathcal{O}\left(\frac{\sigma^2}{2^n}\right)$$

$\mathcal{S}$ makes at most $q_2$ queries and runs in time $\mathcal{O}(q_2 \log q_2)$ where $q_2$ is the number of $\mathsf{Prim}$ queries made by $\mathcal{A}$. ☐

The use of $\mathcal{O}(\cdot)$ just hides small constants. Combining Theorem 4.3 with the indifferentiability composition theorem allows us to conclude security for $\mathrm{HMAC}_d[H]$ for underlying hash function $H$ that is, itself, indifferentiable from a RO. For example, should $H$ be one of the proven-indifferentiable SHA-3 candidates.

**Proof of Theorem 4.3:** **(Sketch)** Fix an allowed key set $\mathcal{K}$. The simulator $\mathcal{S}$ imitates a RO $P\colon \{0,1\}^* \to \{0,1\}^n$ in a way that is consistent with the keyed RO $\mathcal{R}$, to which the simulator has oracle access. The simulator works as shown below.

```
algorithm S^R(U):
Parse U as X || Y with |X| = d
If IsOuter(X) = 0 then
    V ←$ {0,1}^n
    F[V] ← (GetKey(X), Y)
    Ret V
If IsOuter(X) = 1 and F[Y] ≠ ⊥ then
    Z ← R(F[Y])
    Ret Z
Ret R ←$ {0,1}^n
```

In words, the simulator identifies whether a query is associated with an "inner" application or an "outer" application. In the first case it chooses a random response, and records in a table the input associated to the response. If an "outer" application, it looks up in the table whether there is an "inner" input associated with the query. If so, it responds with the keyed RO's output for the key and message associated with that "inner" query. Otherwise, the simulator outputs a random point.

That keys queried to the construction are in an allowed set means that there is no ambiguity in the $\mathsf{IsOuter}$ predicate. Because of this, intuitively, the simulator can only fail in two ways. First, a collision amongst the choices of $V$ across two different queries occur. Second, a query to the simulator with $\mathsf{IsOuter}(X) = 1$ ends up producing a value $V$ such that $V$ was previously queried to the simulator that was an "outer" query. In either cases, the distinguisher can abuse the events to successfully distinguish. Informally, in either the $\mathrm{Real}_{\mathrm{HMAC}_d[P]}$ or game $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}$ the probability of a collision occurring is at most $(q_1+q_2)^2/2^n$ and the probability of the second kind of failure is at most $q_2^2/2^n$. ∎

HMAC USING MD-BASED HASH FUNCTIONS. The above result does not extend to cover HMAC built from hash functions which are not indifferentiable from a RO. This includes, for example, the SHA family of hash functions and others that use the Merkle-Damgård transform. We therefore treat this special case, investigating $\mathrm{HMAC}_d[\mathrm{SMD}[f]]$ where $f\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$ is a random oracle.

**Theorem 4.4** Fix $d, n > 0$ with $d \geq n$. Let $f\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$ be a RO and consider $\mathcal{K}$-restricted $\mathrm{HMAC}_d[\mathrm{SMD}[f]]$ for an allowed key set $\mathcal{K}$. Let $\mathcal{R}\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^n$ be a keyed RO. Then there exists a simulator $\mathcal{S}$ such that for any distinguisher $\mathcal{A}$ whose total query cost is $\sigma \leq 2^{n-2}$ it holds that

$$\mathbf{Adv}^{\mathrm{indiff}}_{\mathrm{HMAC}_d[\mathrm{SMD}[f]],\mathcal{R},\mathcal{S}}(\mathcal{A}) \leq \mathcal{O}\left(\frac{\sigma^2}{2^n}\right)$$

$\mathcal{S}$ makes at most $q_2$ queries and runs in time $\mathcal{O}(q_2 \log q_2)$ where $q_2$ is the number of $\mathsf{Prim}$ queries by $\mathcal{A}$. □

We note that the restriction to $\sigma \leq 2^{n-2}$ in the theorem statement is just a technicality to make the bound simpler and likewise the use of $\mathcal{O}(\cdot)$ in the advantage statement hides just a small constant.

Unlike our positive results about $H^2$, the bounds provided by Theorems 4.3 and 4.4 match, up to small constants, results for other now-standard indifferentiable constructions (c.f., [14]). First, the advantage bounds both hold up to the birthday bound, namely $\sigma \approx 2^{n/2}$. Second, the simulators are efficient and, specifically, make at most one query per invocation. All this enables use of the indifferentiability composition theorem in a way that yields strong, standard concrete security bounds.

For the proof of Theorem 4.4, we simplify the treatment in two ways. First, let $\pi$ be the minimal padding length of SMD. For SHA-256, for example, $\pi = 65$. When $d \geq n + \pi$, the outer application of SMD$[f]$ of HMAC$_d$[SMD$[f]$] will always consist of exactly two $f$ applications. Otherwise, the outer application of SMD$[f]$ could consist of some larger fixed number of $f$ applications. We will for simplicity focus on the case of two calls; it is easy to extend to the more general case.

Second, we will replace the padding of SMD with adversarially-controlled message bits. For any $K \in \mathcal{K}$, any $X \in (\{0,1\}^d)^+$, and any $P \in \{0,1\}^{d-n}$ define the functions

$$\begin{aligned} \mathsf{F}[f](K,X,P) &= \big(f(IV, K \oplus \mathsf{opad})\,,\ \mathrm{MD}[f]((K \oplus \mathsf{ipad}) \,\|\, X)\ \|\ P\big) \\ \mathsf{fF}[f](K,X,P) &= f\big(\mathsf{F}[f](K,X,P)\big) \end{aligned}$$

Here the last bits of $X$ replaces the padding in the first SMD application while $P$ replaces the padding used in the second SMD application. Theorem 4.4 establishes the indifferentiability of $\mathsf{fF}[f]$. In it, the total query cost of an attacker is equal to the sum of the costs (as defined in Section 2) of each query to $\mathsf{Func}$ plus the number of $\mathsf{Prim}$ queries.

**Proof of Theorem 4.4:** Let $g\colon\ \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$ be a RO and let $\mathsf{gF}[f,g](K,X,P) = g\big(\mathsf{F}[f](K,X,P)\big)$. Our proof proceeds in two steps. First, Lemma 4.5 below shows that we can restrict attention to showing indifferentiability for the construction $\mathsf{gF}[f,g]$. This step fundamentally relies upon the fact that keys queried by $\mathcal{A}$ are within an allowed set, which in turn enables us to show domain separation between "internal" uses of $f$ within $\mathsf{F}[f]$ and the external application of $f$ in $\mathsf{fF}[f]$. Second, Lemma 4.6 shows that $\mathsf{F}[f](K,X,P)$ is preimage-aware. This step relies upon the fact that we can apply $\mathsf{GetKey}$ to extract $K$ from $K \oplus \mathsf{ipad}$ and from $K \oplus \mathsf{opad}$. To conclude we combine the two lemmas with [19, Th. 4.1], which asserts that the composition of a PrA function and a RO is indifferentiable from a RO.

**Lemma 4.5** Let $f, g\colon\ \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$ be random oracles. Let $\mathcal{S}_\mathcal{B}$ be a simulator and $\mathcal{A}$ be a distinguisher making at most $q_1$ $\mathsf{Func}$ queries, $q_2$ $\mathsf{Prim}$ queries, and whose total query cost is $\sigma$. Then there exists a simulator $\mathcal{S}_\mathcal{A}$ and adversary $\mathcal{B}$ such that

$$\mathbf{Adv}^{\text{indiff}}_{\mathsf{fF}[f],\mathcal{S}_\mathcal{A}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{indiff}}_{\mathsf{gF}[f,g],\mathcal{S}_\mathcal{B}}(\mathcal{B}) + \frac{10\sigma^2}{2^n}$$

$\mathcal{S}_\mathcal{A}$ makes the same number of queries as $\mathcal{S}_\mathcal{B}$ and works in the same time as $\mathcal{S}_\mathcal{B}$ plus $\mathcal{O}(q_2)$. Adversary $\mathcal{B}$ runs in time that of $\mathcal{A}$ plus $\mathcal{O}(q_1 + q_2)$, makes the same number of queries as $\mathcal{A}$, and has the same total query cost. $\square$

**Proof:** Let $\mathcal{S}_\mathcal{B}$ be an arbitrary simulator for $\mathsf{gF}^{f,g}$. It can be queried on chaining variable, message block pairs $(V, M)$ on either an $f$ interface or $g$ interface. We use the notation $(0, V, M)$ to signal a query to the $f$ interface and $(1, V, M)$ to signify a query to the $g$ interface. We likewise extend the $\mathsf{Prim}$ interface to accept a label for $f$ queries or $g$ queries.

Let $\mathcal{A}$ be an indifferentiability adversary against $\mathsf{fF}$.

| | |
|---|---|
| $\underline{\mathcal{S}_{\mathcal{A}}^{\mathcal{R}}(V, M)}$<br>If $\mathtt{T}[V] = 1$ then Ret $\mathcal{S}_{\mathcal{B}}^{\mathcal{R}}(1, V, M)$<br>$V' \leftarrow \mathcal{S}_{\mathcal{B}}^{\mathcal{R}}(0, IV, M)$<br>If $(V = IV \wedge \mathsf{IsOuter}(M) = 1)$ then<br>    $\mathtt{T}[V'] \leftarrow 1$<br>Ret $V'$ | $\underline{\mathcal{B}^{\mathsf{Func},\mathsf{Prim}}}$<br>$b' \leftarrow\$ \mathcal{A}^{\mathsf{Func}_a,\mathsf{Prim}_a}$<br>Ret $b'$<br><br>**procedure** $\underline{\mathsf{Func}_a(K, X, P)}$:<br>Ret $Y \leftarrow \mathsf{Func}(K, X, P)$<br><br>**procedure** $\underline{\mathsf{Prim}_a(V, M)}$:<br>If $\mathtt{T}[V] = 1$ then Ret $\mathsf{Prim}(1, V, M)$<br>$V' \leftarrow \mathsf{Prim}(0, IV, M)$<br>If $(V = IV \wedge \mathsf{IsOuter}(M) = 1)$ then<br>    $\mathtt{T}[V'] \leftarrow 1$<br>    Ret $V'$ |

Figure 11: Simulator $\mathcal{S}_{\mathcal{A}}$ and adversary $\mathcal{B}$ used in proof of Lemma 4.5.

We construct a simulator $\mathcal{S}_{\mathcal{A}}$ and an adversary $\mathcal{B}$ as shown in Figure 11. These keep track of the responses $V'$ to $f$-queries on $IV, M$ with $\mathsf{IsOuter}(M) = 1$ using a table $\mathtt{T}$. A subsequent query made with chaining variable that equals any such response is treated as a $g$-query. In this way, $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{B}$ "route" queries to either an $f$ or $g$ interface in an appropriate way.

By construction $\text{Ideal}^{\mathcal{A}}_{\mathcal{R},\mathcal{S}_{\mathcal{A}}}$ and $\text{Ideal}^{\mathcal{B}}_{\mathcal{R},\mathcal{S}_{\mathcal{B}}}$ are equivalent.

We now use a sequence of games to argue bound the difference between $\Pr[\text{Real}^{\mathcal{A}}_{\mathsf{fF}[f]} \Rightarrow 1]$ and $\Pr[\text{Real}^{\mathcal{B}}_{\mathsf{gF}[f,g]} \Rightarrow 1]$. See Figure 12. Game $\text{G}_0^{\mathcal{A}}$ implements exactly $\text{Real}^{\mathcal{A}}_{\mathsf{fF}[f]}$ — the extra book-keeping code in $\mathsf{Prim}_a$ does not affect its behavior. The procedure $\mathsf{R}_x$ is used to implement the RO $f$; the variable 0 is used as a label. Game $\text{G}_1$ includes the boxed pseudocode, which changes $\text{G}_0$ in that now random choices of chaining variables $V$ are restricted so as to not collide with any chaining variable so-far seen in the game (including $IV$). Recall that $\sigma$ is the maximum number of blocks of message bits queried by $\mathcal{A}$, and this is thus the total number of $\mathsf{R}_x$ invocations. Over the course of the game $|\mathcal{V}| \leq 2\sigma$ and so a standard birthday bound argument combined with the discussion above justifies that

$$\Pr\left[\, \text{Real}^{\mathcal{A}}_{\mathsf{fF}[f]} \Rightarrow 1 \,\right] \;=\; \Pr\left[\, \text{G}_0^{\mathcal{A}} \Rightarrow 1 \,\right] \leq \Pr\left[\, \text{G}_1^{\mathcal{A}} \Rightarrow 1 \,\right] + \frac{(2\sigma)^2}{2^n} \;.$$

The next game $\text{G}_2$ is equivalent to $\text{G}_1$, except that we label random choices in $\mathsf{R}$ by either 0 or 1. These labels are added as entries to the table, but lines 31 and 32 ensure an entry already made for a domain point with a different labeling is used by a subsequent call. Thus,

$$\Pr\left[\, \text{G}_2 \Rightarrow 1 \,\right] = \Pr\left[\, \text{G}_1 \Rightarrow 1 \,\right] \;.$$

Game $\text{G}_3$ is the same as $\text{G}_2$ but with the boxed statement removed. By the fundamental lemma of game-playing [10] we have that

$$\Pr\left[\, \text{G}_2 \Rightarrow 1 \,\right] \leq \Pr\left[\, \text{G}_3 \Rightarrow 1 \,\right] + \Pr\left[\, \text{G}_3 \text{ sets } \mathsf{bad} \,\right] \;.$$

We now argue that only a call to $\mathsf{R}$ on line 21 has non-zero probability of setting $\mathsf{bad}$. We then bound the probability of line 21 causing $\mathsf{bad}$ to be set. For the first step, we find it helpful to cast the state of the game's random choices as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ whose nodes are $n$-bit strings and whose edges are $d$-bit strings. When a new entry $V' = \mathtt{R}[x, V, X]$ is chosen, add both $V, V'$ to $\mathcal{V}$ if they are not already present in the graph, and add the edge $(V, V')$ with label $X$. We additionally color $V'$ by $x$. Should $V'$ later be returned by $\mathsf{R}(x, V, X)$ and $\mathtt{R}[1-x, V, X] \neq \perp$, then we recolor $V'$ to $x$. Because of

**main** $\mathrm{G}_0$ $\boxed{\mathrm{G}_1}$:

$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Func}_a,\mathsf{Prim}_a}$
Ret $b'$

**procedure** $\mathsf{Func}_a(K, X, P)$:

$V_i \leftarrow \mathrm{MD}[\mathsf{R}(0,\cdot,\cdot)](K \oplus \mathsf{ipad} \,\|\, X)$
$V_o \leftarrow \mathsf{R}(0, IV, K \oplus \mathsf{opad})$
Ret $Y \leftarrow \mathsf{R}(0, V_o, V_i \,\|\, P)$

**procedure** $\mathsf{Prim}_a(V, M)$:

If $\mathtt{T}[V] = 1$ then Ret $\mathsf{R}(0, V, M)$
$V' \leftarrow \mathsf{R}(0, V, M)$
If $(V = IV \wedge \mathsf{IsOuter}(M) = 1)$ then
  $\mathtt{T}[V'] \leftarrow 1$
Ret $V'$

**subroutine** $\mathsf{R}(x, V, M)$

$\mathcal{V} \xleftarrow{\cup} V$ ; $V' \leftarrow\!\!\$\ \{0,1\}^n$ $\boxed{\setminus \mathcal{V}}$
If $\mathtt{R}[x, V, M] \neq \bot$ then
  $V' \leftarrow \mathtt{R}[x, V, M]$
$\mathcal{V} \xleftarrow{\cup} V'$
Ret $\mathtt{R}[x, V, M] \leftarrow V'$

**main** $\boxed{\mathrm{G}_2}$ $\mathrm{G}_3$:

00 $b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Func}_a,\mathsf{Prim}_a}$
01 Ret $b'$

**procedure** $\mathsf{Func}_a(K, X, P)$:

10 $V_i \leftarrow \mathrm{MD}[\mathsf{R}(0,\cdot,\cdot)](K \oplus \mathsf{ipad} \,\|\, X)$
11 $V_o \leftarrow \mathsf{R}(0, IV, K \oplus \mathsf{opad})$
12 Ret $Y \leftarrow \mathsf{R}(1, V_o, V_i \,\|\, P)$

**procedure** $\mathsf{Prim}_a(V, M)$:

20 If $\mathtt{T}[V] = 1$ then Ret $\mathsf{R}(1, V, M)$
21 $V' \leftarrow \mathsf{R}(0, V, M)$
22 If $(V = IV \wedge \mathsf{IsOuter}(M) = 1)$ then
23   $\mathtt{T}[V'] \leftarrow 1$
24 Ret $V'$

**subroutine** $\mathsf{R}(x, V, M)$

30 $\mathcal{V} \xleftarrow{\cup} V$ ; $V' \leftarrow\!\!\$\ \{0,1\}^n \setminus \mathcal{V}$
31 If $\mathtt{R}[1-x, V, M] \neq \bot$ then
32   $\mathsf{bad} \leftarrow \mathsf{true}$ $\boxed{; x \leftarrow 1 - x}$
33 If $\mathtt{R}[x, V, M] \neq \bot$ then
34   $V' \leftarrow \mathtt{R}[x, V, M]$
35 $\mathcal{V} \xleftarrow{\cup} V'$
36 Ret $\mathtt{R}[x, V, M] \leftarrow V'$

**main** $\boxed{\mathrm{G}_4}$ $\mathrm{G}_5$:

$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Func}_a,\mathsf{Prim}_a}$
Ret $b'$

**procedure** $\mathsf{Func}_a(K, X, P)$:

$V_i \leftarrow \mathrm{MD}[\mathsf{R}(0,\cdot,\cdot)](K \oplus \mathsf{ipad} \,\|\, X)$
$V_o \leftarrow \mathsf{R}(0, IV, K \oplus \mathsf{opad})$
$\mathcal{X} \leftarrow V_o$
Ret $Y \leftarrow \mathsf{R}(1, V_o, V_i \,\|\, P)$

**procedure** $\mathsf{Prim}_a(V, M)$:

If $\mathtt{T}[V] = 1$ then Ret $\mathsf{R}(1, V, M)$
If $V \in \mathcal{X}$ then $\mathsf{bad} \leftarrow \mathsf{true}$
$V' \leftarrow \mathsf{R}(0, V, M)$
If $(V = IV \wedge \mathsf{IsOuter}(M) = 1)$ then
  $\mathtt{T}[V'] \leftarrow 1$
Ret $V'$

**subroutine** $\mathsf{R}(x, V, M)$

$\mathcal{V} \xleftarrow{\cup} V$ ; $V' \leftarrow\!\!\$\ \{0,1\}^n$ $\boxed{; \setminus \mathcal{V}}$
If $\mathtt{R}[x, V, M] \neq \bot$ then
  $V' \leftarrow \mathtt{R}[x, V, M]$
$\mathcal{V} \xleftarrow{\cup} V'$
Ret $\mathtt{R}[x, V, M] \leftarrow V'$

Figure 12: Games used in the proof of Lemma 4.5.

the game's restrictions on the choice of $V'$, we have that $\mathcal{G}$ is throughout the game a forest. The root of one tree is $IV$ and all others are rooted at adversarially-chosen values $V$ (corresponding to queries to $\mathsf{Prim}_a$).

We now argue by case analysis that the probability of recoloring is zero for any call to $\mathsf{R}$ but one made on line 21. Recall that we disallow pointless queries, meaning the adversary never queries $\mathsf{Func}_a$ or $\mathsf{Prim}_a$ twice on the same values.

- Line 10: A recoloring here means that at the time of the query a path $IV, V_1, V_2, \ldots, V_k$ for some number $k$ exists in $\mathcal{G}$ with edge $(IV, V_1)$ labeled by $X = K \oplus \mathsf{ipad}$ for the queried $K$ and the color of node $V_k$ was 1. But the only such paths that can exist in $\mathcal{G}$ must have $(IV, V_1)$ labeled by an $X$ with $\mathsf{IsOuter}(X) = 1$. But this contradicts that $K \in \mathcal{K}$.
- Line 11: No paths of length one rooted at $IV$ can exist in $\mathcal{G}$ with the second node colored 1.
- Line 12: A recoloring here means that at the time of the query a path $IV, V_1, V_2$ exists in $\mathcal{G}$ with edge $(IV, V_1)$ labeled by $X = K \oplus \mathsf{opad}$ for the queried $K$ and $V_3$ is colored 0. If the path was formed due to a $\mathsf{Func}_a$ query, then this must mean that $X = K' \oplus \mathsf{ipad}$ for some other $K'$ and this contradicts that $K, K' \in \mathcal{K}$. If the path was formed due to $\mathsf{Prim}_a$ queries, then the check on line 22 means that the $\mathsf{IsOuter}(X) = 0$, but this contradicts that $K \in \mathcal{K}$.
- Line 20: A recoloring here means that at the time of the query a path $IV, V_1, V_2$ exists in $\mathcal{G}$ with edge $(IV, V_1)$ labeled by $X$ with $\mathsf{IsOuter}(X) = 1$ (by the check on line 22) and $V_3$ colored by 0.

But the only way such a path can exist is due to execution of line 10 with $X = K \oplus \mathsf{ipad}$ for the value $K$ of this prior query. This contradicts that $K \in \mathcal{K}$.

The flag $\mathsf{bad}$ in $\mathrm{G}_3$ can only be set due to a query on line 21, and so game $\mathrm{G}_4$ makes this explicit by moving the setting of $\mathsf{bad}$ to $\mathsf{Prim}_a$. Also, we make the setting of $\mathsf{bad}$ more liberal by only tracking chaining variable values. These changes have no effect on the values returned to the adversary. We have that

$$\Pr[\,\mathrm{G}_3 \Rightarrow 1\,] = \Pr[\,\mathrm{G}_4 \Rightarrow 1\,] \quad \text{and} \quad \Pr[\,\mathrm{G}_3 \text{ sets } \mathsf{bad}\,] \le \Pr[\,\mathrm{G}_4 \text{ sets } \mathsf{bad}\,] \ .$$

We now bound the setting of $\mathsf{bad}$ in $\mathrm{G}_4$. The flag is set due to a query $(V, M)$ to $\mathsf{Prim}_a$ such that: (1) $V \in \mathcal{X}$, meaning $V = V_o$ for the latter chosen in $\mathsf{Func}_a$ as the result of $V_o \leftarrow \mathsf{R}_0(IV, K \oplus \mathsf{opad})$; and (2) $\mathtt{T}[V] \neq 1$, meaning that no previous query to $\mathsf{Prim}_a$ was made on $IV, M'$ with $\mathsf{IsOuter}(M') = 1$. But together, (1) and (2) imply that the value $V \in \mathcal{X}$ has not yet been returned to the adversary at the time of the query. This means that the adversary has no knowledge of the coins underlying the choice of $V$ and can narrow it down only by the fact that $V$ is not equal to any other chaining variable returned. Thus, the probability that the adversary can query $V$ is at most $1/(2^n - |\mathcal{V}|)$ where $|\mathcal{V}|$ is the size of $\mathcal{V}$ at the time of the query. Taking a union bound over all queries to $\mathsf{Prim}_a$ and using the facts that $|\mathcal{V}| \le 2\sigma$ and $|\mathcal{X}| \le q_1$ we have that

$$\Pr[\,\mathrm{G}_4 \text{ sets } \mathsf{bad}\,] \le \frac{q_1 q_2}{2^n - 2\sigma} \le \frac{2\sigma^2}{2^n}$$

and where we have additionally used our restriction that $\sigma \le 2^{n-2}$.

Finally, game $\mathrm{G}_5$ relaxes the restrictions on selection of chaining variables. A birthday-bound argument establishes that

$$\Pr[\,\mathrm{G}_4 \Rightarrow 1\,] \le \Pr[\,\mathrm{G}_5 \Rightarrow 1\,] + \frac{(2\sigma)^2}{2^n} \ .$$

Combining the above equations gives the bound claimed in the lemma. ▮

The next lemma shows that $\mathsf{F}[f]$ is preimage-aware [19]. See Section 2 for the formal definition of preimage-awareness.

**Lemma 4.6** Let $f$ be a random oracle $f\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^n$. Then there exists an extractor $\mathcal{E}$ such that for any adversary $\mathcal{A}$ making $q_e$ extraction queries, at most $q$ $f$-queries, and outputs a message of total length at most $d\ell$. Then it holds that

$$\mathbf{Adv}^{\mathrm{pra}}_{\mathsf{F}[f],\mathcal{E}}(\mathcal{A}) \le \frac{q_e \ell (q+\ell)}{2^n} + \frac{(q+\ell)^2}{2^n} + \frac{q}{2^n} \ .$$

□

We just sketch the proof, which is straightforward. Recall that

$$\mathsf{F}[f](K, X, P) = f(IV, K \oplus \mathsf{ipad}) \ \| \ \mathrm{MD}[f]((K \oplus \mathsf{opad}) \,\|\, X) \ \| \ P$$

We observe that it is trivial to extract $P$. For $K$, one can investigate queries of the form $f(IV, M)$ for $\mathsf{IsOuter}(X) = 1$ and determine $K$ via $\mathsf{GetKey}(M)$. Remaining is extracting $X$, but this follows from the preimage awareness of $\mathrm{MD}[f]$, which is implied by combining [19, Th. B.1] and [19, Th. 3.2]. ▮

## Acknowledgments

The authors thank Hugo Krawczyk for providing significant feedback and suggestions, in particular encouraging the authors to include positive results for the indifferentiability of HMAC; Niels Ferguson for in-depth discussions regarding the security of $H^2$; and the anonymous reviewers for their helpful suggestions. Dodis was supported in part by NSF grants CNS-1065134, CNS-1065288, CNS-1017471, CNS-0831299. Ristenpart was supported in part by NSF grant CNS-1065134. Steinberger is supported by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61061130540, 61073174, and by NSF grant 0994380. Tessaro was supported in part by NSF grants CCF-0915675, CCF-1018064, and DARPA contracts FA8750-11-C-0096, FA8750-11-2-0225.

## References

[1] Elena Andreeva, Bart Mennink, and Bart Preneel. On the indifferentiability of the Grøstl hash function. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10: 7th International Conference on Security in Communication Networks*, volume 6280 of *Lecture Notes in Computer Science*, pages 88–105. Springer, September 2010.

[2] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 130–146. Springer, December 2007.

[3] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, August 2006.

[4] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 232–249. Springer, December 2009.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, August 1996.

[6] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: RKA-PRPs, RKA-PRFs, and applications. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 491–506. Springer, May 2003.

[7] Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology – ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer, December 2006.

[8] Mihir Bellare and Thomas Ristenpart. Hash functions in the dedicated-key setting: Design choices and MPP transforms. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP 2007: 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 399–410. Springer, July 2007.

[9] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *Advances in Cryptology – CRYPTO ‘12*, Lecture Notes in Computer Science. Springer, 2012.

[10] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, May / June 2006.

[11] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 1–18. Springer, April 2003.

[12] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, April 2008.

[13] Donghoon Chang and Mridul Nandi. Improved indifferentiability security analysis of chopMD hash function. In Kaisa Nyberg, editor, *Fast Software Encryption – FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 429–443. Springer, February 2008.

[14] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, August 2005.

[15] Dana Dachman-Soled, Rosario Gennaro, Hugo Krawczyk, and Tal Malkin. Computational extractors and pseudorandomness. In Ronald Cramer, editor, *Theory of Cryptography – TCC ’12*, volume 7194 of *Lecture Notes in Computer Science*, pages 383–403. Springer, 2012.

[16] Ivan Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO’89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, August 1990.

[17] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 494–510. Springer, August 2004.

[18] Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to MD6. In Orr Dunkelman, editor, *Fast Software Encryption – FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 104–121. Springer, February 2009.

[19] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. Salvaging Merkle-Damgård for practical applications. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 371–388. Springer, April 2009.

[20] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO’92*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, August 1993.

[21] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer, August 2005.

[22] Niels Ferguson and Bruce Schneier. *Practical cryptography.* Wiley, 2003.

[23] Pierre-Alain Fouque, David Pointcheval, and Sébastien Zimmer. HMAC is a randomness extractor and applications to TLS. In Masayuki Abe and Virgil Gligor, editors, *ASIACCS 08: 3rd Conference on Computer and Communications Security*, pages 21–32. ACM Press, March 2008.

[24] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, and L. Stewart. An Extension to HTTP: Digest Access Authentication. RFC 2069 (Proposed Standard), January 1997. Obsoleted by RFC 2617.

[25] Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). IETF RFC 2409 (Proposed Standard), 1998.

[26] Shoichi Hirose, Je Hong Park, and Aaram Yun. A simple variant of the Merkle-Damgård scheme with a permutation. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 113–129. Springer, December 2007.

[27] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *ISOC Network and Distributed System Security Symposium – NDSS'99*. The Internet Society, February 1999.

[28] Ghassan Karame and Srdjan Capkun. Low-cost client puzzles based on modular exponentiation. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS 2010: 15th European Symposium on Research in Computer Security*, volume 6345 of *Lecture Notes in Computer Science*, pages 679–697. Springer, 2010.

[29] Charlie Kaufman. The Internet Key Exchange (IKEv2) Protocol. IETF RFC 4306 (Proposed Standard), 2005.

[30] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[31] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869 (Proposed Standard), January 2010.

[32] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 631–648. Springer, August 2010.

[33] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, February 2004.

[34] Ralph C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, August 1990.

[35] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. A modular security analysis of the TLS handshake protocol. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 55–73. Springer, December 2008.

[36] PKCS #5: Password-based cryptography standard (rfc 2898). RSA Data Security, Inc., September 2000. Version 2.0.

[37] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, May 2011.

[38] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Network and Distributed Systems Security – NDSS '10*. ISOC, 2010.

[39] Douglas Stebila, Lakshmi Kuppusamy, Jothi Rangasamy, Colin Boyd, and Juan Manuel González Nieto. Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 284–301. Springer, February 2011.

[40] Gene Tsudik. Message authentication with one-way hash functions. In *Proceedings IEEE INFOCOM'92*, volume 3, pages 2055–2059. IEEE, 1992.

[41] XiaoFeng Wang and Michael K. Reiter. Defending against denial-of-service attacks with puzzle auction. In *IEEE Symposium on Security and Privacy*, pages 78–92, 2003.

[42] Scott Yilek. Resettable public-key encryption: How to encrypt on a virtual machine. In Josef Pieprzyk, editor, *Topics in Cryptology – CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 41–56. Springer, March 2010.

# A Internal Collision Probabilities

We briefly discuss computing the probabilities $p(H, w, \ell)$ and $p'(H, w, \ell)$ for the case where $H[P] = P = \mathcal{R}$.

To compute $p(H, w, \ell)$ for $H^2 = H^2[P]$, let $z_0$ be a randomly chosen $n$-bit string, and define $z_i$ such that $z_i = \mathcal{R}(z_{i-1})$ for all $i = 1, \ldots, 2\ell + 1$. Note that in particular $z_0, z_1, \ldots, z_{2\ell}, z_{2\ell+1}$ correspond to the values $x_0, y_0, \ldots, x_\ell, y_\ell$ in the definition of $p(H, w, \ell)$. We first upper bound the probability of the event $\mathsf{bad}_1$ that there is a collision among the $z$ values, which is

$$\begin{aligned}
\Pr[\,\mathsf{bad}\,] &\leq \sum_{i=0}^{2\ell+1} \Pr[\,z_i \in \{z_0, z_1, \ldots, z_{i-1}\} \mid |\{z_0, z_1, \ldots, z_{i-1}\}| = i\,] \\
&= \sum_{i=0}^{2\ell+1} \frac{i}{2^n} \leq \frac{(2\ell+1)(2\ell+2)}{2 \cdot 2^n} \leq \frac{2 \cdot (\ell+1)^2}{2^n} \,.
\end{aligned}$$

Conditioned on $\overline{\mathsf{bad}}$, we have $x_\ell \notin \{y_0, y_1, \ldots, y_\ell\}$. Moreover, let $\mathsf{bad}_2$ be the event that for some

| | |
|---|---|
| **adversary** $\mathcal{D}_{w,\ell}^{\mathsf{Func},\mathsf{Prim}}$: | **adversary** $\widetilde{\mathcal{D}}_{w,l}^{\mathsf{Func},\mathsf{Prim}}$: |
| $\mathbf{u}[1],\ldots,\mathbf{u}[w] \leftarrow\!\!\$\; \{0,1\}^n$ | $\mathbf{u}[1],\ldots,\mathbf{u}[w] \leftarrow\!\!\$\; \{0,1\}^n$ |
| $x_0 \leftarrow\!\!\$\; \{0,1\}^n$ ; $j \leftarrow\!\!\$\; [1\,..\,w]$ | $x_0 \leftarrow\!\!\$\; \{0,1\}^n$ ; $j \leftarrow\!\!\$\; [1\,..\,w]$ |
| For $i = 1$ to $\ell$ do | For $i = 1$ to $\ell$ do |
| $\quad x_i \leftarrow \mathsf{Func}(x_{i-1})$ | $\quad x_i \leftarrow \mathsf{Func}(0^d, x_{i-1})$ |
| $\mathbf{u}[j] \leftarrow x_\ell$ | $\mathbf{u}[j] \leftarrow x_\ell$ |
| For $i = 1$ to $w$ do | For $i = 1$ to $w$ do |
| $\quad \mathbf{v}[i] \leftarrow\!\!\$\; H[\mathsf{Prim}](\mathbf{u}[i])$ | $\quad \mathbf{v}[i] \leftarrow\!\!\$\; H[\mathsf{Prim}](\mathsf{ipad} \parallel \mathbf{u}[j])$ |
| $y_0 \leftarrow\!\!\$\; H[\mathsf{Prim}](x_0)$ | $y_0 \leftarrow\!\!\$\; H[\mathsf{Prim}](\mathsf{ipad} \parallel x_0)$ |
| For $i = 1$ to $\ell$ do | For $i = 1$ to $\ell$ do |
| $\quad y_i \leftarrow\!\!\$\; \mathsf{Func}(y_{i-1})$ | $\quad y_i \leftarrow\!\!\$\; \mathsf{Func}(\mathsf{opad} \oplus \mathsf{ipad}, y_{i-1})$ |
| Ret $(y_\ell = \mathbf{v}[j]) \wedge (x_\ell \notin \{y_0, y_1, \ldots, y_\ell\})$ $\quad \wedge(\forall i \neq j : \mathbf{v}[i] \notin \{y_0, y_1, \ldots, y_\ell\})$. | Ret $(y_\ell = \mathbf{v}[j]) \wedge(\forall i \neq j : \mathbf{v}[i] \notin \{y_0, y_1, \ldots, y_\ell\})$. |

Figure 13: Left: Adversary $\mathcal{D}_{w,\ell}$ used in the proof of Theorem 3.2. Right: Adversary $\widetilde{\mathcal{D}}_{w,\ell}$ used in the proof of Theorem 4.2. The notation $H[\mathsf{Prim}](x)$ indicates the evaluation of the hash construction $H$ on input $x$ in which $P$ queries are replied by the corresponding $\mathsf{Prim}$ queries.

$i \in [1\,..\,w-1]$ we get $\mathcal{R}(U_i) \in \{y_0, y_1, \ldots, y_\ell\}$. Then,

$$\begin{aligned}\Pr\left[\,\mathsf{bad}_2 \mid \overline{\mathsf{bad}_1}\,\right] &\le \sum_{i\neq j} \Pr\left[\,U_i \in \{x_0, x_1, \ldots x_\ell\} \mid \overline{\mathsf{bad}_1}\,\right] \\ &\qquad + \sum_{i \neq j} \Pr\left[\,\mathcal{R}(U_i) \in \{y_0, y_1, \ldots, y_\ell\} \mid U_i \notin \{x_0, x_1, \ldots, x_\ell\} \wedge \overline{\mathsf{bad}_1}\,\right] \\ &= \frac{2(w-1)\cdot(\ell+2)}{2^n}\;.\end{aligned}$$

Therefore, $p(\mathcal{R}, w, \ell) \le \Pr\left[\,\mathsf{bad}_1\,\right] + \Pr\left[\,\mathsf{bad}_2 \mid \overline{\mathsf{bad}_1}\,\right] \le \frac{2(w-1)\cdot(\ell+2)}{2^n} + \frac{2\cdot(\ell+1)^2}{2^n}$.

It is not hard to see that the same upper bound can be computed for $p'$, where the $z$-values are the intermediate values with respect to $\mathcal{R}$ and HMAC[$\mathcal{R}$].

# B Proof of Theorem 3.2

A formal description of the adversary $\mathcal{D}_{w,\ell}$ sketched above is provided in Figure 13.

As the first step, note that by the chain-shift property of $H^2[P]$, the condition $(y_\ell = \mathbf{v}[j])$ is always satisfied in the real-world experiment. Consequently,

$$\Pr\left[\,\mathrm{Real}_{H^2[P]}^{\mathcal{D}_{w,\ell}} \Rightarrow 1\,\right] \ge 1 - p(H, w, \ell)\;.$$

The remainder of this proof consists of upper bounding the probability $\Pr\left[\,\mathrm{Ideal}_{\mathrm{RO},\mathcal{S}[\mathrm{RO}]}^{\mathcal{D}_{w,\ell}} \Rightarrow 1\,\right]$ under the constraint that the simulator makes $q_{\mathcal{S}}$ queries. To this end, it is convenient to introduce a security game – which we refer to as G and is described in Figure 14– involving a (stateful) adversary $\mathcal{B}$ and a random oracle RO. The adversary $\mathcal{B}$ is given a vector $\mathbf{u}$ of $w$ $n$-bit strings. $n-1$ of these are chosen uniformly at random and independently. The remaining one, placed in a randomly chosen component $j \in [1\,..\,w]$, is the output $x_\ell$ of a chain of $\ell$ random-oracle invocations starting at a randomly chosen initial $n$-bit string $x_0$. The adversary is asked to output a vector $\mathbf{v}$ of $w$ *distinct* $n$-bit values, to which

```
procedure main: // G
phase ← 1
u[1], ..., u[w] ←$ {0,1}^n
x_0 ←$ {0,1}^n ; j ←$ [1 .. w]
For i = 1 to ℓ do
    x_i ←$ RO(x_{i-1})
u[j] ← x_ℓ
v ←$ B^RO(u) ; phase ← 2 ; y_0 ← B^RO(x_0)
For i = 1 to ℓ do
    y_i ←$ RO(y_{i-1})
Ret (y_ℓ = v[j]) ∧ (x_ℓ ∉ {y_0, y_1, ..., y_ℓ})
        ∧ (∀i ≠ j : v[i] ∉ {y_0, y_1, ..., y_ℓ})

procedure RO(x): // G, G_0 - G_2, G_5
If R[x] = ⊥ then
    R[x] ←$ {0,1}^n
Ret R[x]
```

```
procedure main: // G_0
phase ← 1
u[1], ..., u[w] ←$ {0,1}^n
x_0 ←$ {0,1}^n ; j ←$ [1 .. w]
For i = 1 to ℓ do
    x_i ←$ RO(x_{i-1})
u[j] ← x_ℓ
v ←$ B^RO(u) ; phase ← 2
y_0 ← B^RO(x_0)
Ret (FC(j) ≠ ∅)

subroutine FC(i) // G_0 − G_7
P ← ∅
For all (y'_0, y_1,' ..., y'_ℓ) do
    If (phase = 2) ∧ (y'_ℓ = v[i])
        ∧ (∀k ∈ [1 .. ℓ] : R[y'_{k-1}] = y'_k)
        ∧ (∀i' ≠ i : v[i'] ∉ {y'_0, y'_1, ..., y'_ℓ})
        ∧ (x_ℓ ∉ {y'_0, y'_1, ..., y'_ℓ}) then
        P ←∪ {(y_0, y_1, ..., y_ℓ)}
Ret P
```

Figure 14: Games G and $G_0$. Note that the adversary $\mathcal{B}$ keeps a state across its first and second invocations.

it commits. At this point, $x_0$ is revealed to $\mathcal{B}$ (and hence, indirectly, also $j$), which now needs to output a value $y_0$ such that the output $y_\ell$ of an chain of RO invocations starting at $y_0$ equals $\mathbf{v}[j]$, and such that $x_\ell$, as well as $\mathbf{v}[i]$ for $i \neq j$, is not part of this chain.

The following lemma establishes the rather direct relation between the task of building a simulator for $H^2[P]$ and providing a good adversary $\mathcal{B}$ for the game G.

**Lemma B.1** For all simulators $\mathcal{S}$ making $q_{\mathcal{S}}$ queries, there exists an adversary $\mathcal{B}$ making $q' = q_{\mathcal{S}} + \ell$ queries such that
$$\Pr\left[\, \mathrm{Ideal}_{\mathcal{R},\mathcal{S}}^{\mathcal{D}_{w,\ell}} \Rightarrow 1 \,\right] = \Pr\left[\, G^{\mathcal{B}} \Rightarrow 1 \,\right] .$$
Moreover, whenever $\mathcal{B}$ outputs $y_0$, then it has issued all RO queries to compute the chain $y_0, y_1, \ldots, y_\ell$ starting in $y_0$.

**Proof of of Lemma B.1:** The adversary $\mathcal{B}$, upon receiving the vector $\mathbf{u}$, runs an execution of the simulator $\mathcal{S}$, feeding it with $\mathsf{Prim}$ queries $\mathbf{u}[1], \ldots, \mathbf{u}[m]$, and obtaining replies $\mathbf{v}[1], \ldots, \mathbf{v}[m]$. It outputs $\mathbf{v}$ to conclude the first phase of G. In particular, RO queries by the simulator are replied directly by the oracle in the game G. Then, upon receiving $x_0$, $\mathcal{B}$ continues the execution of the simulator (recall that $\mathcal{B}$ is sateful), asking $\mathsf{Prim}$ queries to $\mathcal{S}$ in order to compute $y_0 = H[P](x_0)$. It continues by making all RO queries to evaluate the chain $y_0, y_1, \ldots, y_\ell$ such that $y_i = \mathrm{RO}(y_{i-1})$. Finally, $\mathcal{B}$ outputs $y_0$. It is not hard to verify that the probability that $\mathcal{B}$ wins the game is exactly $\Pr\left[\, \mathrm{Ideal}_{\mathcal{R},\mathcal{S}}^{\mathcal{D}_{w,\ell}} \Rightarrow 1 \,\right]$. ∎

| **procedure** main: // $\boxed{\mathrm{G}_1}$, $\mathrm{G}_2$ |
|---|
| phase $\leftarrow 1$ |
| $\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow_{\$} \{0,1\}^n$ |
| $x_0 \leftarrow_{\$} \{0,1\}^n$ ; $j \leftarrow_{\$} [1 \,..\, w]$ |
| For $i = 1$ to $\ell - 1$ do |
|     $x_i \leftarrow_{\$} \mathrm{RO}(x_{i-1})$ |
| If $\mathtt{R}[x_{\ell-1}] = \perp$ then |
|     $\mathtt{R}[x_{\ell-1}] \leftarrow \mathbf{u}[j]$ |
| Else |
|     bad $\leftarrow$ true ; $\boxed{\mathbf{u}[j] \leftarrow \mathtt{R}[x_{\ell-1}]}$ |
| $x_\ell \leftarrow \mathbf{u}[j]$ |
| $\mathbf{v} \leftarrow_{\$} \mathcal{B}^{\mathrm{RO}}(\mathbf{u})$ ; phase $\leftarrow 2$ |
| $y_0 \leftarrow \mathcal{B}^{\mathrm{RO}}(x_0)$ |
| Ret $(\mathsf{FC}(j) \neq \emptyset)$ |

| **procedure** main: // $\boxed{\mathrm{G}_3}$, $\mathrm{G}_4$ |
|---|
| phase $\leftarrow 1$ |
| $\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow_{\$} \{0,1\}^n$ |
| $x_0 \leftarrow_{\$} \{0,1\}^n$ ; $j \leftarrow_{\$} [1 \,..\, w]$ |
| For $i = 1$ to $\ell - 1$ do |
|     $x_i \leftarrow_{\$} \mathrm{RO}(x_{i-1})$ |
| $\boxed{\text{If } \mathtt{R}[x_{\ell-1}] = \perp \text{ then } \mathtt{R}[x_{\ell-1}] \leftarrow \mathbf{u}[j]}$ |
| $x_\ell \leftarrow \mathbf{u}[j]$ |
| $\mathbf{v} \leftarrow_{\$} \mathcal{B}^{\mathrm{RO}}(\mathbf{u})$ ; phase $\leftarrow 2$ |
| If $\mathtt{R}[x_{\ell-1}] = \perp$ then $\mathtt{R}[x_{\ell-1}] \leftarrow \mathbf{u}[j]$ |
| $y_0 \leftarrow \mathcal{B}^{\mathrm{RO}}(x_0)$ |
| Ret $(\mathsf{FC}(j) \neq \emptyset)$ |
| |
| **subroutine** $\mathrm{RO}(x)$: // $\mathrm{G}_3$, $\mathrm{G}_4$ |
| If $\mathtt{R}[x] = \perp$ then |
|     If $(x = x_{\ell-1}) \wedge (\text{phase} = 1)$ then |
|         bad $\leftarrow$ true |
|     $\mathtt{R}[x] \leftarrow_{\$} \{0,1\}^n$ |
| Ret $\mathtt{R}[x]$ |

Figure 15: Pseudocode descriptions of Games $\mathrm{G}_1$ - $\mathrm{G}_4$.

In the following, we focus on showing an upper bound on $\Pr\left[\, G^{\mathcal{B}} \Rightarrow 1 \,\right]$ for a $q'$-query adversary $\mathcal{B}$ which, without loss of generality, outputs a value $y_0$ such that it has asked all RO queries defining a chain $(y_0, y_1, \ldots, y_\ell)$ starting in $y_0$. As a first step, we consider the game $\mathrm{G}_0$, depicted in Figure 14, which is similar to G, but possibly slightly easier to win. In $\mathrm{G}_0$, we introduce a sub-routine, called $\mathsf{FC}$, which on input $i$ returns the sets of tuples $(y'_0, y'_1, \ldots, y'_\ell)$ such that, with respect to RO queries asked so far, define a chain such that $y'_\ell = \mathbf{v}[i]$ and $y'_i = \mathrm{RO}(y'_{i-1})$ for $i = 1, \ldots, \ell$, and $x_\ell$ and $\mathbf{v}[i]$ for $i \neq j$ is not part of this chain. In particular, if $\mathbf{v}$ has not been defined yet (i.e., phase $= 1$), then $\mathsf{FC}$ always returns the empty set. We also modify the winning condition so that it returns true as long as there is a chain $(y'_0, y'_1, \ldots, y'_\ell) \in \mathsf{FC}(j)$ (i.e., it does not need to be the one starting at the value $y_0$ output by $\mathcal{B}$). Then, clearly

$$\Pr\left[\, G^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \leq \Pr\left[\, {\mathrm{G}_0}^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \;,$$

as $\mathcal{B}$ winning in G implies $\mathcal{B}$ winning $\mathrm{G}_0$ because of the fact that $\mathcal{B}$ has asked all queries corresponding to the chain starting at the output string $y_0$.

We continue with Game $\mathrm{G}_1$ (cf. Figure 15) which is equivalent to $\mathrm{G}_0$. The (merely syntactical) difference is that it starts by first choosing all components of $\mathbf{u}$ uniformly at random. It then compute the chain $x_0, x_1, \ldots$ only up to $x_{\ell-1}$, and then sets $\mathtt{R}[x_{\ell-1}]$ to equal $\mathbf{u}[j]$ if it is undefined, whereas otherwise it overwrites $\mathbf{u}[j]$ as $\mathbf{u}[j] \leftarrow \mathtt{R}[x_{\ell-1}]$, and sets the flag bad. In both cases, $x_\ell$ equals $\mathbf{u}[j]$. The next Game $\mathrm{G}_2$ (also in Figure 15) is then obtained from $\mathrm{G}_1$ by modifying the latter case so that the value $\mathbf{u}[j]$ is *not* overwritten. Clearly, $\mathrm{G}_1$ and $\mathrm{G}_2$ are equivalent until bad. With $\mathbf{Adv}(\mathrm{G}_1^{\mathcal{B}}, \mathrm{G}_2^{\mathcal{B}}) =$

```
procedure main: // G5, [G6]
phase ← 1
u[1], ..., u[w] ←$ {0,1}^n
v ←$ B^RO(u) ; phase ← 2
x0 ←$ {0,1}^n ; j ←$ [1..w] ; x_ℓ ← u[j]
For i = 1 to ℓ − 1 do
    x_i ←$ RO(x_{i−1})
If R[x_{ℓ−1}] = ⊥ then R[x_{ℓ−1}] ← u[j]
[If (FC(j) ≠ ∅) ∧ (FC′(j) = ∅) then bad ← true]
x_ℓ ← u[j]
y0 ← B^RO(x0)
Ret (FC(j) ≠ ∅)
```

```
procedure main: // G7
phase ← 1
u[1], ..., u[w] ←$ {0,1}^n
v ←$ B^RO(u) ; phase ← 2
x0 ←$ {0,1}^n ; j ←$ [1..w] ; x_ℓ ← u[j]
For i = 1 to ℓ − 1 do
    x_i ←$ RO(x_{i−1})
If R[x_{ℓ−1}] = ⊥ then R[x_{ℓ−1}] ← u[j]
If (FC(j) ≠ ∅) ∧ (FC′(j) = ∅) then bad ← true
y0 ← B^RO(x0)
Ret (FC′(j) ≠ ∅)
```

```
subroutine FC′(i): // G6 − G7
P ← ∅
For all (y′0, y1,′ ..., y′ℓ) do
    If (phase = 2) ∧ (y′ℓ = v[i])
        ∧ (∀k ∈ [1..ℓ] : R′[y′_{k−1}] = y′_k)
        ∧ (∀i′ ≠ i : v[i′] ∉ {y′0, y′1, ..., y′ℓ})
        ∧ (x_ℓ ∉ {y′0, y′1, ..., y′ℓ}) then
        P ←∪ {(y0, y1, ..., yℓ)}
Ret P
```

```
procedure RO(x): // G6 - G7
If R[x] = ⊥ then
    R[x] ←$ {0,1}^n
    If phase = 1 then R′[x] ← R[x]
    If (phase = 2) ∧ (FC(j) ≠ ∅) ∧ (FC′(j) = ∅)
then
        bad ← true
Ret R[x]
```

Figure 16: Pseudocode descriptions of Games $G_5$ - $G_7$.

$\Pr\left[\, G_1^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] - \Pr\left[\, G_2^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right]$, by the fundamental lemma of game playing we obtain

$$\mathbf{Adv}(G_1^{\mathcal{B}}, G_2^{\mathcal{B}}) \leq \Pr\left[\, G_1^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] \leq \sum_{i=0}^{\ell-2} (i+1) \cdot 2^{-n} \leq \frac{\ell^2}{2^{n+1}} \ ,$$

as conditioned on $x_0, \ldots, x_i$ being distinct, each output $\mathrm{RO}(x_i)$ for $i \in [0 \,..\, \ell-2]$ is in $\{x_0, \ldots, x_i\}$ with probability $(i+1)/2^n$.

Note that in $G_2$, the input $\mathbf{u}$ to $\mathcal{B}$ is random and independent of everything else, as long as $\mathcal{B}$ does not query $x_{\ell-1}$ to RO. We first transition from $G_2$ into new games $G_3$ and $G_4$ (also on Figure 15). In $G_3$, the game sets bad if the query $\mathrm{RO}(x_{\ell-1})$ is made before $\mathcal{B}$ commits to $\mathbf{v}$. Clearly, $\mathbf{Adv}(G_2^{\mathcal{B}}, G_3^{\mathcal{B}}) = 0$. Additionally, $G_4$ is modified so that the value $\mathtt{R}[x_{\ell-1}]$ is set to equal $\mathbf{u}[j]$ only *after* $\mathcal{B}$ has output $\mathbf{v}$. Hence, $G_3$ and $G_4$ are equivalent until bad, and

$$\mathbf{Adv}(G_3^{\mathcal{B}}, G_4^{\mathcal{B}}) \leq \Pr\left[\, G_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] \ .$$

We postpone an analysis of $\Pr\left[\, G_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right]$ to a later point in the proof, and now continue with the main sequence of games.

The next game, Game $G_5$ (cf. Figure 16), simply rearranges the contents of Game $G_4$ for better readability, but is otherwise fully equivalent. In particular, we have now postponed the computation of the values $x_1, \ldots, x_\ell$ to after $\mathcal{B}$ outputs $\mathbf{v}$, which clearly does not affect the game. Then, we transition to a game $G_6$ which takes into account (via a second procedure $\mathsf{FC}'$) which chains have been created before the adversary outputs $\mathbf{v}$, and after this, sets the condition $\mathsf{bad}$ as soon as some new $\mathtt{R}[\cdot]$ entry is defined such that $\mathsf{FC}'(j) = \emptyset$ but $\mathsf{FC}(j) \neq \emptyset$ for the chosen $j$. Finally, $G_7$ is the same as $G_6$, but the winning condition checks for $(y'_0, y'_1, \ldots, y'_\ell) \in \mathsf{FC}'(j)$. Clearly, $G_6$ and $G_7$ are equivalent until $\mathsf{bad}$, and thus

$$\mathbf{Adv}(G_6^{\mathcal{B}}, G_7^{\mathcal{B}}) \leq \Pr\left[\, G_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] .$$

Finally, combining all transitions, we obtain

$$\Pr\left[\, G^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \leq \frac{\ell^2}{2^{n+1}} + \Pr\left[\, G_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] + \Pr\left[\, G_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] + \Pr\left[\, G_7^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] .$$

To conclude the proof, we now turn to upper bounding the three probabilities on the RHS.

Upper bounding $\Pr\left[\, G_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right]$. By inspection, it is not hard to verify that the probability that $\mathsf{bad}$ is set in $G_4$ equals the probability that $\mathcal{B}$ wins the following game $G'_4$:

```
procedure main: // G'_4
Q ← ∅ ; u[1],...,u[w] ←$ {0,1}^n
v ←$ B^RO(u)
x_0 ←$ {0,1}^n ; j ←$ [1..w]
For i = 1 to ℓ−1 do
    If R[x_{i−1}] = ⊥ then R[x_{i−1}] ←$ {0,1}^n
    x_i ← R[x_{i−1}]
Ret x_{ℓ−1} ∈ Q
```

```
procedure RO(x): // G'_4
Q ←∪ {x}
If R[x] = ⊥ then
    R[x] ←$ {0,1}^n
Ret R[x]
```

This is because first, we can focus on phase 1, and second, we can postpone the generation of the values $x_0, x_1, \ldots, x_{\ell-1}$ to after the execution of the adversary $\mathcal{B}$. To upper bound the probability, let $\mathsf{bad}_i$ be the event that of $\{x_0, x_1, \ldots, x_i\} \cap Q \neq \emptyset$. Then,

$$\begin{aligned}
\Pr\left[\, G_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] &= \Pr\left[\, {G'_4}^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \\
&\leq \Pr\left[\, \mathsf{bad}_{\ell-1} \,\right] \\
&\leq \Pr\left[\, \mathsf{bad}_0 \,\right] + \sum_{i=1}^{\ell-1} \Pr\left[\, \mathsf{bad}_i \mid \overline{\mathsf{bad}_{i-1}} \,\right] \leq \frac{\ell \cdot q'}{2^n} ,
\end{aligned}$$

since $\Pr\left[\, \mathsf{bad}_0 \,\right] = q'/2^n$, and conditioned on $\mathsf{bad}_{i-1}$ not having occurred, we have two cases when $x_{i-1}$ is defined: (i) $\mathtt{R}[x_{i-1}] = \bot$. Here, $x_i = \mathrm{RO}(x_{i-1}) = \mathtt{R}[x_{i-1}]$ is chosen uniformly, and hence it collides with one of the values in $Q$ with probability at most $q'/2^n$. (ii) $\mathtt{R}[x_{i-1}] \neq \bot$. But then, this means that $x_{i-1} = x_j$ for $j < i-1$, as $x_j \notin Q$ by $\overline{\mathsf{bad}_{i-1}}$. In turn, it must also be that $\mathtt{R}[x_{i-1}] = \mathtt{R}[x_j] = x_{j+1} \notin Q$ as $\overline{\mathsf{bad}_{i-1}}$ holds, and thus the probability of provoking $\overline{\mathsf{bad}_i}$ is 0.

Upper bounding $\Pr\left[\, G_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right]$. We seek for an upper bound on the probability that a RO query $x$ in the second phase of the game $G_7$ provokes $\mathsf{FC}(j) \neq \emptyset$ when $\mathsf{FC}'(j) = \emptyset$. First, note that it can never be that setting $\mathtt{R}[x_{\ell-1}]$ to $\mathbf{u}[j]$ provokes $\mathsf{FC}(j)$ to become non-empty: Every new chain $(y'_0, y'_1, \ldots, y'_\ell)$ being defined thanks to $\mathtt{R}[x_{\ell-1}]$ being set to $\mathbf{u}[j] = x_\ell$ contains $x_\ell$, and hence is not returned by $\mathsf{FC}(j)$. Therefore, a new chain $(y'_0, y'_1, \ldots, y'_q)$ must be defined when $\mathtt{R}[x]$ is set to a fresh random value, for some string $x$.

Let $Z$ be the set of strings $z_0 \in \{0,1\}^n$ such that there exists $z_1, \ldots, z_{\ell'}$ with $\ell' < \ell$, $z_{\ell'} = \mathbf{v}[j]$, and $\mathtt{R}'[z_{i-1}] = z_i$ for all $i = 1, \ldots, \ell'$. Then, note that as long as no random choice of $\mathtt{R}[x]$ in phase 2 hits one element of $Z$, the set $\mathsf{FC}(j)$ remains empty. The probability that one such values hits $Z$ is clearly at most $(q' \cdot |Z|)/2^n$. Since $|Z| \leq q'$, then this means

$$\Pr\left[\, \mathrm{G}_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] \leq q' \cdot 2^{-n} .$$

UPPER BOUNDING $\Pr\left[\, \mathrm{G}_7^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right]$. Assume that $\mathcal{B}$ indeed outputs a vector $\mathbf{v}$ with $w$ distinct components after making at most $q'$ queries. Note that any two chains $(y'_0, y'_1, \ldots, y'_\ell) \in \mathsf{FC}'(i')$ and $(y''_0, y''_1, \ldots, y''_\ell) \in \mathsf{FC}'(i'')$ are disjoint. As each chain is built by means of asking $\ell$ RO-queries, this means that there are at most $\lfloor \frac{q_{\mathcal{S}}}{\ell} \rfloor$ indices $j$ for which $\mathsf{FC}'(j)$ is not empty. Hence,

$$\Pr\left[\, \mathrm{G}_7^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \leq \frac{1}{w}\frac{q'}{\ell} ,$$

as the choice of $j$ is independent of the behavior of the adversary $\mathcal{B}$ so far. ∎

# C Proof of Theorem 4.2

Let $H = H[P]$ be the hash function with $n$-bit output, and let HMAC$[H]$ be the corresponding instantiation of HMAC using $H$. A description of the adversary $\widetilde{\mathcal{D}}_{w,\ell}$ is given in Figure 13, on the right. The attacker first chooses a random $n$-bit string $x_0$, and then builds a chain of values of $(x_0, x_1, \ldots, x_\ell)$ so that $x_i = \mathsf{Func}(0^d, x_{i-1})$ for all $i \in [1\mathinner{..}\ell]$. It then chooses a random index $j \in [1\mathinner{..}w]$, and sets $\mathbf{u}[j] \leftarrow x_\ell$, whereas $\mathbf{u}[i]$ is set to a random value for all $i \neq j$. The next step has $\widetilde{\mathcal{D}}_{w,\ell}$ asking all necessary $\mathsf{Prim}$ queries to evaluate $H$ on input $\mathsf{ipad} \parallel \mathbf{u}[i]$ for all $i \in [1\mathinner{..}w]$, obtaining outputs $\mathbf{v}[1], \ldots, \mathbf{v}[w]$. Subsequently, $\widetilde{\mathcal{D}}_{w,\ell}$ also makes all $\mathsf{Prim}$ queries needed to evaluate $H$ on input $\mathsf{ipad} \parallel x_0$, and we refer to resulting output $n$-bit string as $y_0$. Finally, $\widetilde{\mathcal{D}}_{w,\ell}$ computes $y_1, \ldots, y_\ell$, where $y_i = \mathsf{Func}(\mathsf{opad} \oplus \mathsf{ipad}, y_{i-1})$ for all $i \in [1\mathinner{..}\ell]$, and outputs 1 if and only if $y_\ell = \mathbf{v}[j]$ (it outputs 0 otherwise).

Let us start by observing that in the real-world experiment $\mathrm{Real}_{\mathrm{HMAC}[H]}^{\widetilde{\mathcal{D}}_{w,\ell}}$, by the chain-shift property of HMAC, we will always have $y_\ell = \mathbf{v}[j]$. Therefore, the probability that the attacker $\widetilde{\mathcal{D}}_{w,\ell}$ outputs 0 is bounded by the probability that there exists $i \neq j$ such that $\mathbf{v}[i] \in \{y_0, y_1, \ldots, y_\ell\}$. The probability that this happens is, by definition, exactly $p'(H, w, \ell)$, and thus

$$\Pr\left[\mathrm{Real}_{\mathrm{HMAC}[H]}^{\widetilde{\mathcal{D}}_{w,\ell}} \Rightarrow 1\right] = 1 - p'(H, w, \ell) .$$

The remainder of this proof analyzes the ideal world experiment $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}^{\widetilde{\mathcal{D}}_{w,\ell}}$. As a first step, we proceed by showing a reduction to a new game, denote $\widetilde{\mathrm{G}}$, which we define in Figure 17. This is essentially a version of Game G considered earlier in Figure 14, but using two independent random oracles $\mathrm{RO}_1$ and $\mathrm{RO}_2$ instead.

**Lemma C.1** For all simulators $\mathcal{S}$ making $q_{\mathcal{S}}$ queries, there exists an adversary $\mathcal{B}$ making $q' = q_{\mathcal{S}} + \ell$ queries such that

$$\Pr\left[\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}^{\widetilde{\mathcal{D}}_{w,\ell}} \Rightarrow 1\right] = \Pr\left[\, \widetilde{\mathrm{G}}^{\mathcal{B}} \Rightarrow 1 \,\right] .$$

Moreover, whenever $\mathcal{B}$ outputs $y_0$, then it has already issued all $\mathrm{RO}_2$ queries to compute the chain $y_0, y_1, \ldots, y_\ell$.

**Proof of of Lemma C.1:** The adversary $\mathcal{B}$ internally simulates the simulator $\mathcal{S}$. Simulator random oracle queries of the form $(\mathsf{ipad}, x)$ are answered as $\mathrm{RO}_1(x)$, queries of the form $(\mathsf{opad}, x)$ are answered

**procedure** main: // $\widetilde{\mathrm{G}}$

phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow\$ \{0,1\}^n$
$x_0 \leftarrow\$ \{0,1\}^n$ ; $j \leftarrow\$ [1 \mathinner{..} w]$
For $i = 1$ to $\ell$ do
  $x_i \leftarrow\$ \mathrm{RO}_1(x_{i-1})$
$\mathbf{u}[j] \leftarrow x_\ell$
$\mathbf{v} \leftarrow\$ \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(x_0)$
For $i = 1$ to $\ell$
  $y_i \leftarrow\$ \mathrm{RO}_2(y_{i-1})$
Ret $(y_\ell = \mathbf{v}[j]) \wedge (\forall i \neq j : \mathbf{v}[i] \notin \{y_0, y_1, \ldots, y_\ell\})$.

**procedure** main: // $\widetilde{\mathrm{G}}_0$

phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow\$ \{0,1\}^n$
$x_0 \leftarrow\$ \{0,1\}^n$ ; $j \leftarrow\$ [1 \mathinner{..} w]$
For $i = 1$ to $\ell$ do
  $x_i \leftarrow\$ \mathrm{RO}_1(x_{i-1})$
$\mathbf{u}[j] \leftarrow x_\ell$
$\mathbf{v} \leftarrow\$ \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,RO_2}(x_0)$
Ret $(\mathsf{FC}(j) \neq \emptyset)$

**procedure** $\mathrm{RO}_i(x)$: // $\widetilde{\mathrm{G}}$, $\widetilde{\mathrm{G}}_0$ - $\widetilde{\mathrm{G}}_2$, $\widetilde{\mathrm{G}}_5$

If $\mathtt{R}_i[x] = \perp$ then
  $\mathtt{R}_i[x] \leftarrow\$ \{0,1\}^n$
Ret $\mathtt{R}_i[x]$

**subroutine** $\mathsf{FC}(i)$ // $\widetilde{\mathrm{G}}_0 - \widetilde{\mathrm{G}}_7$

$P \leftarrow \emptyset$
For all $(y_0', y_1,' \ldots, y_\ell')$ do
  If (phase $= 2$) $\wedge$ $(y_\ell' = \mathbf{v}[i])$
    $\wedge (\forall i \neq j : \mathbf{v}[i] \notin \{y_0, y_1, \ldots, y_\ell\})$
    $\wedge$ $(\forall k \in [1 \mathinner{..} \ell] : \mathtt{R}_2[y_{k-1}'] = y_k')$ then
    $P \xleftarrow{\cup} \{(y_0', y_1', \ldots, y_\ell')\}$
Ret $P$

**procedure** main: // $\boxed{\widetilde{\mathrm{G}}_1}$, $\widetilde{\mathrm{G}}_2$

phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow\$ \{0,1\}^n$
$x_0 \leftarrow\$ \{0,1\}^n$ ; $j \leftarrow\$ [1 \mathinner{..} w]$
For $i = 1$ to $\ell - 1$ do
  $x_i \leftarrow\$ \mathrm{RO}_1(x_{i-1})$
If $\mathtt{R}_1[x_{\ell-1}] = \perp$ then
  $\mathtt{R}_1[x_{\ell-1}] \leftarrow \mathbf{u}[j]$
Else
  $\mathsf{bad} \leftarrow \mathsf{true}$ ; $\boxed{\mathbf{u}[j] \leftarrow \mathtt{R}_1[x_{\ell-1}]}$
$x_\ell \leftarrow \mathbf{u}[j]$
$\mathbf{v} \leftarrow\$ \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(x_0)$
Ret $(\mathsf{FC}(j) \neq \emptyset)$

**procedure** main: // $\boxed{\widetilde{\mathrm{G}}_3}$, $\widetilde{\mathrm{G}}_4$

phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow\$ \{0,1\}^n$
$x_0 \leftarrow\$ \{0,1\}^n$ ; $j \leftarrow\$ [1 \mathinner{..} w]$
For $i = 1$ to $\ell - 1$ do
  $x_i \leftarrow\$ \mathrm{RO}_1(x_{i-1})$
$\boxed{\text{If } \mathtt{R}_1[x_{\ell-1}] = \perp \text{ then } \mathtt{R}_1[x_{\ell-1}] \leftarrow \mathbf{u}[j]}$
$x_\ell \leftarrow \mathbf{u}[j]$
$\mathbf{v} \leftarrow\$ \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
If $\mathtt{R}_1[x_{\ell-1}] = \perp$ then $\mathtt{R}_1[x_{\ell-1}] \leftarrow \mathbf{u}[j]$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(x_0)$
Ret $(\mathsf{FC}(j) \neq \emptyset)$

**subroutine** $\mathrm{RO}_i(x)$: // $\widetilde{\mathrm{G}}_3$, $\widetilde{\mathrm{G}}_4$

If $\mathtt{R}_i[x] = \perp$ then
  If $(i = 1) \wedge (x = x_{\ell-1}) \wedge (\text{phase} = 1)$ then
    $\mathsf{bad} \leftarrow \mathsf{true}$
  $\mathtt{R}_i[x] \leftarrow\$ \{0,1\}^n$
Ret $\mathtt{R}_i[x]$

Figure 17: Games $\widetilde{\mathrm{G}}$ and $\widetilde{\mathrm{G}}_1$ - $\widetilde{\mathrm{G}}_4$. Note that the adversary $\mathcal{B}$ keeps a state across its first and second invocations.

as $\mathrm{RO}_2(x)$, whereas $\mathcal{B}$ internally simulates answers to all other types of RO queries. Then, when receiving the vector $\mathbf{u}$, the adversary $\mathcal{B}$ makes $\mathsf{Prim}$ queries to $\mathcal{S}$ to evaluate $H(\mathsf{ipad} \,\|\, \mathbf{u}[i])$ for all $i$, and outputting the results $\mathbf{v}[1], \ldots, \mathbf{v}[w]$ of these evaluations as a vector. Then, upon receiving $x_0$, $\mathcal{B}$ asks $\mathsf{Prim}$ queries to $\mathcal{S}$ to evaluate $H$ on input $\mathsf{ipad} \,\|\, x_0$, and once the result $y_0$ is computed, first makes all $\mathrm{RO}_2$ queries to evaluate the chain of length $\ell$ starting at $y_0$ of length $\ell$, and then outputs $y_0$. Clearly, the probability that $\mathcal{B}$ outputs 1 is the same as the probability that $\widetilde{\mathcal{D}}_{w,\ell}$ outputs 1 in $\mathrm{Ideal}_{\mathcal{R},\mathcal{S}}^{\widetilde{\mathcal{D}}_{w,\ell}}$, since $\mathsf{Func}$ queries needed to compute the two chains give independent outputs due to $0^d \neq \mathsf{ipad} \oplus \mathsf{opad}$. ▮

The rest of the proof is similar to the one of Theorem 3.2. Specifically we now focus on upper bounding $\Pr\left[\, \widetilde{\mathrm{G}}^{\mathcal{B}} \Rightarrow 1 \,\right]$ for the adversary $\mathcal{B}$ built from $\mathcal{S}$ as in the lemma statement making $q'$ queries. As a first step, we consider the game $\widetilde{\mathrm{G}}_0$, depicted in Figure 17, which is similar to $\widetilde{\mathrm{G}}$, but is potentially easier to win. Concretely, in $\widetilde{\mathrm{G}}_0$, we introduce a sub-routine, called $\mathsf{FC}$, which on input $i$ returns the sets of tuples $(y'_0, y'_1, \ldots, y'_\ell)$ such that, with respect to $\mathrm{RO}_2$ queries asked so far, define a chain such that $y'_\ell = \mathbf{v}[i]$ and $y'_i = \mathrm{RO}_2(y'_{i-1})$ for $i = 1, \ldots, \ell$, and $\mathbf{v}[i]$ for $i \neq j$ is not part of this chain. In particular, if $\mathbf{v}$ has not been defined yet (i.e., phase $= 1$), then $\mathsf{FC}$ always returns the empty set. We also modify the winning condition so that it returns true as long as there is a chain $(y'_0, y'_1, \ldots, y'_\ell) \in \mathsf{FC}(j)$ (i.e., it does not need to be the one starting at the value $y_0$ output by $\mathcal{B}$). Then, clearly

$$\Pr\left[\, \widetilde{G}^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \leq \Pr\left[\, \widetilde{\mathrm{G}}_0^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] ,$$

as $\mathcal{B}$ winning in $\widetilde{\mathrm{G}}$ implies $\mathcal{B}$ winning $\widetilde{\mathrm{G}}_0$ because of the fact that $\mathcal{B}$ has asked all queries corresponding to the chain starting at $y_0$. We continue with Game $\widetilde{G}_1$, which is equivalent to $\mathrm{G}_0$. The only modification is purely syntactical: The game starts by first choosing all components of $\mathbf{u}$ uniformly at random. It then compute the chain $x_0, x_1, \ldots$ only up to $x_{\ell-1}$, and then sets $\mathtt{R}_1[x_{\ell-1}]$ to equal $\mathbf{u}[j]$ if it is undefined, whereas otherwise it overwrites $\mathbf{u}[j]$ as $\mathbf{u}[j] \leftarrow \mathtt{R}_1[x_{\ell-1}]$, and sets the flag $\mathsf{bad}$. In both cases, $x_\ell$ equals $\mathbf{u}[j]$. The next game, Game $\widetilde{G}_2$, is derived from $\mathrm{G}_1$ by modifying the latter case so that the value $\mathbf{u}[j]$ is *not* overwritten. Clearly, $\widetilde{G}_1$ and $\widetilde{G}_2$ are equivalent until $\mathsf{bad}$, and therefore $\mathbf{Adv}(\widetilde{G}_1^{\mathcal{B}}, \widetilde{G}_2^{\mathcal{B}}) \leq \Pr\left[\, \widetilde{G}_1^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] \leq \sum_{i=0}^{\ell-2}(i+1) \cdot 2^{-n} \leq \frac{\ell^2}{2^{n+1}}$, as conditioned on $x_0, \ldots, x_i$ being distinct, each output $\mathrm{RO}_1(x_i)$ for $i \in [0 \mathinner{..} \ell-2]$ is in $\{x_0, \ldots, x_i\}$ with probability $(i+1)/2^n$.

Note that in $\widetilde{G}_2$, the input $\mathbf{u}$ to $\mathcal{B}$ is random and independent of everything else, as long as $\mathcal{B}$ does not query $x_{\ell-1}$ to $\mathrm{RO}_1$. We transition from $\widetilde{G}_2$ into new games $\widetilde{G}_3$ and $\widetilde{G}_4$. In $\widetilde{G}_3$, the game sets $\mathsf{bad}$ if the query $\mathrm{RO}_1(x_{\ell-1})$ is made before $\mathcal{B}$ commits to $\mathbf{v}$. Clearly, $\mathbf{Adv}(\widetilde{G}_2^{\mathcal{B}}, \widetilde{G}_3^{\mathcal{B}}) = 0$. Additionally, $\widetilde{G}_4$ is modified so that the value $\mathtt{R}_1[x_{\ell-1}]$ is set to equal $\mathbf{u}[j]$ only *after* $\mathcal{B}$ has output $\mathbf{v}$. Hence, $\widetilde{G}_3$ and $\widetilde{G}_4$ are equivalent until $\mathsf{bad}$, and $\mathbf{Adv}(\widetilde{G}_3^{\mathcal{B}}, \widetilde{G}_4^{\mathcal{B}}) \leq \Pr\left[\, \widetilde{G}_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right]$. We will prove an upper bound on this probability below, but for now continue with the main sequence of games.

The next game, Game $\widetilde{G}_5$, simply rearranges the contents of Game $\widetilde{G}_4$ for better readability, but is otherwise fully equivalent. In particular, we postpone the computation of the values $x_1, \ldots, x_\ell$ to after $\mathcal{B}$ outputs $\mathbf{v}$, which clearly does not affect the game. Then, we transition to a game $\widetilde{G}_6$ which takes into account (via a second procedure $\mathsf{FC}'$) those chains that have been created before the adversary outputs $\mathbf{v}$, and after this, sets the condition $\mathsf{bad}$ as soon as some new $\mathtt{R}_2[\cdot]$ entry is defined such that $\mathsf{FC}'(j) = \emptyset$ but $\mathsf{FC}(j) \neq \emptyset$ for the chosen $j$. Finally, $\widetilde{G}_7$ is the same as $\widetilde{G}_6$, but the winning condition checks for $(y'_0, y'_1, \ldots, y'_\ell) \in \mathsf{FC}'(j)$. Clearly, $\widetilde{G}_6$ and $\widetilde{G}_7$ are equivalent until $\mathsf{bad}$, and thus $\mathbf{Adv}(\widetilde{G}_6^{\mathcal{B}}, \widetilde{G}_7^{\mathcal{B}}) \leq \Pr\left[\, \widetilde{G}_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right]$. Finally, combining all transitions, we obtain

$$\Pr\left[\, \widetilde{\mathrm{G}}^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] \leq \frac{\ell^2}{2^{n+1}} + \Pr\left[\, \widetilde{\mathrm{G}}_4^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] + \Pr\left[\, \widetilde{\mathrm{G}}_7^{\mathcal{B}} \text{ sets } \mathsf{bad} \,\right] + \Pr\left[\, \widetilde{\mathrm{G}}_7^{\mathcal{B}} \Rightarrow \mathsf{true} \,\right] .$$

**procedure** main: // $\widetilde{G}_5$, $\boxed{\widetilde{G}_6}$
phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow_{\$} \{0,1\}^n$
$\mathbf{v} \leftarrow_{\$} \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
$x_0 \leftarrow_{\$} \{0,1\}^n$ ; $j \leftarrow_{\$} [1 \mathinner{..} w]$ ; $x_\ell \leftarrow \mathbf{u}[j]$
For $i = 1$ to $\ell - 1$ do
  $x_i \leftarrow_{\$} \mathrm{RO}_1(x_{i-1})$
If $\mathtt{R}_1[x_{\ell-1}] = \perp$ then $\mathtt{R}_1[x_{\ell-1}] \leftarrow \mathbf{u}[j]$
$\boxed{\text{If } (\mathsf{FC}(j) \neq \emptyset) \wedge (\mathsf{FC}'(j) = \emptyset) \text{ then } \mathsf{bad} \leftarrow \mathsf{true}}$
$x_\ell \leftarrow \mathbf{u}[j]$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(x_0)$
Ret $(\mathsf{FC}(j) \neq \emptyset)$

**procedure** main: // $\widetilde{G}_7$
phase $\leftarrow 1$
$\mathbf{u}[1], \ldots, \mathbf{u}[w] \leftarrow_{\$} \{0,1\}^n$
$\mathbf{v} \leftarrow_{\$} \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(\mathbf{u})$
phase $\leftarrow 2$
$x_0 \leftarrow_{\$} \{0,1\}^n$ ; $j \leftarrow_{\$} [1 \mathinner{..} w]$ ; $x_\ell \leftarrow \mathbf{u}[j]$
For $i = 1$ to $\ell - 1$ do
  $x_i \leftarrow_{\$} \mathrm{RO}_1(x_{i-1})$
If $\mathtt{R}[x_{\ell-1}] = \perp$ then $\mathtt{R}_1[x_{\ell-1}] \leftarrow \mathbf{u}[j]$
If $(\mathsf{FC}(j) \neq \emptyset) \wedge (\mathsf{FC}'(j) = \emptyset)$ then $\mathsf{bad} \leftarrow \mathsf{true}$
$y_0 \leftarrow \mathcal{B}^{\mathrm{RO}_1,\mathrm{RO}_2}(x_0)$
Ret $(\mathsf{FC}'(j) \neq \emptyset)$

**subroutine** $\mathsf{FC}'(i)$: // $\widetilde{G}_6 - \widetilde{G}_7$
$P \leftarrow \emptyset$
For all $(y'_0, y_1,' \ldots, y'_\ell)$ do
  If (phase $= 2$) $\wedge$ $(y'_\ell = \mathbf{v}[i])$
    $\wedge\ (\forall k \in [1 \mathinner{..} \ell] : \mathtt{R}'_2[y'_{k-1}] = y'_k)$
    $\wedge\ (\forall i' \neq i : \mathbf{v}[i'] \notin \{y'_0, y'_1, \ldots, y'_\ell\})$ then
    $P \xleftarrow{\cup} \{(y_0, y_1, \ldots, y_\ell)\}$
Ret $P$

**procedure** $\mathrm{RO}_2(x)$: // $\widetilde{G}_6$ - $\widetilde{G}_7$
If $\mathtt{R}_2[x] = \perp$ then
  $\mathtt{R}_2[x] \leftarrow_{\$} \{0,1\}^n$
  If phase $= 1$ then $\mathtt{R}'_2[x] \leftarrow \mathtt{R}[x]$
  If (phase $= 2$) $\wedge$ $(\mathsf{FC}(j) \neq \emptyset)$ $\wedge$ $(\mathsf{FC}'(j) = \emptyset)$
then
    $\mathsf{bad} \leftarrow \mathsf{true}$
Ret $\mathtt{R}[x]$

Figure 18: Descriptions of Games $\widetilde{G}_5$ - $\widetilde{G}_7$.

To conclude the proof, we need to upper bound the three probabilities on the RHS. This is very similar to the proof of Theorem 3.2, and is omitted.

# D Proof of Theorem 3.3

The simulator $\mathcal{S}$ that we use is given by the procedure OnRightQuery$(\cdot)$ in game G0 of Figure 19. The simulator maintains four tables: $\mathtt{g}$, $\mathtt{g}^{-1}$, $\mathtt{G}$ and $\mathtt{G}^{-1}$. These tables, as well as other arrays in subsequent games, are assumed to have their entries initialized to $\perp$ at the start of the game. We note these tables can be interpreted as four seperate directed graphs of vertex set $\{0,1\}^n$, or as a single directed graph of vertex set $\{0,1\}^n$ with edges of four different colors.

We use sans-serif variable names such as $\mathsf{bad}$, $\mathsf{fresh}$ and $\mathsf{KnownToSim}$ for boolean values. All boolean values are implicitly initialized to $\mathsf{false}$. Notation such as $\mathtt{gProxy}[x].\mathsf{fresh}$ (see e.g. game $\mathrm{G}_{11}$) indicates that each entry in the table $\mathtt{gProxy}$ holds a bit-value $\mathsf{fresh}$ besides the $n$-bit value $\mathtt{gProxy}[x]$ itself. We emphasize that $\mathsf{fresh}$ is not an attribute of the *value* $\mathtt{gProxy}[x]$, but just an additional bit of data stored at the $x$-th entry of the table $\mathtt{gProxy}[]$. Such bits are also assumed to be initialized to $\mathsf{false}$ at the start of the game.

It will be convenient to view the internal randomness of each game (to be distinguished from $\mathcal{A}$'s randomness) as a tape divided into $n$-bits *blocks*. Thus, the instruction $x \leftarrow_{\$} \{0,1\}^n$ should be viewed as setting $x$ equal to the next unread block of the random tape.

The boolean value bad is called a *flag*. Once set to true, bad is never reset to false. We recall the standard fact that if two games $\mathrm{G}_i, \mathrm{G}_{i+1}$ differ only for instructions that occur after the flag bad is set to true (see for example games $\mathrm{G}_2$, $\mathrm{G}_3$) then

$$|\Pr[\mathcal{A}^{\mathrm{G}_i} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathrm{G}_{i+1}} \Rightarrow 1]| \leq \Pr[\mathcal{A}^{\mathrm{G}_j} \text{ sets } \mathsf{bad} \leftarrow \mathsf{true}].$$

for either $j = i$ or $j = i+1$, where all the probabilities are computed both over $\mathcal{A}$'s coins and over the random coins used in each game.

The *query history* of an adversary $\mathcal{A}$ after its $i$-th query has been answered is the sequence $\mathbf{Q}_i = (Q_j)_{j=1}^{i}$, where each $Q_j$ is a tuple in $\{0,1\}^n \times \{0,1\}^n \times \{\mathtt{l}, \mathtt{r}\}$. For example, if $Q_j = (x_j, y_j, \mathtt{l})$, this indicates $\mathcal{A}$'s $j$-th query was $x_j\{0,1\}^n$, asked to its left oracle, and that it received the value $y_j \in \{0,1\}^n$ in response. Also, $\mathcal{A}$'s *query sequence* $\mathbf{Q}_i^*$ of first $i$ queries is $\big((x_j, *_j)\big)_{j=1}^{i}$ if $\mathbf{Q}_i = \big((x_j, y_j, *_j)\big)_{j=1}^{i}$. Namely, the query sequence records which queries were asked, without the answers.

We note that the game $\mathrm{G}_0$ (Figure 19) is equivalent to the oracle pair $(\mathrm{RO}, \mathcal{S})$ and that game $\mathrm{G}_{23}$ is equivalent to the oracle pair $(G[g], g)$ for a random $g$. As written, the simulator given by OnRightQuery$(\cdot)$ in game $\mathrm{G}_0$ makes more than $3q_1 + 1$ calls to RO, however many of these are redundant, and could be eliminated by having the simulator check its table G before calling RO. It is not difficult to see that once redundant calls are eliminated the simulator never makes more than $3q_1 + 1$ calls to RO per query it answers.

We proceed to upper bound $\mathcal{A}$'s distinguising advantage $\Delta(\mathrm{G}_i, \mathrm{G}_{i+1}) := |\Pr[\mathcal{A}^{\mathrm{G}_i} \Rightarrow 1] - \Pr[\mathcal{A}^{\mathrm{G}_{i+1}} \Rightarrow 1]|$ between games $\mathrm{G}_i, \mathrm{G}_{i+1}$ for $0 \leq i < 21$.

**($\mathbf{G_0} \rightarrow \mathbf{G_1}$; Figure 19.)** Game $\mathrm{G}_1$ implements the RO via lazy sampling, taking place in the subroutine ROsub. Moreover, all random sampling is done via the subroutine Random(), and the "simulator" (which is still, roughly speaking, the procedure OnRightQuery(), though such a distinction will progressively become harder to make) make its queries to ROsub() via the subroutine SimROsub(). In particular, this means that $\mathtt{G}^{-1}$ is set only for queries made by the simulator, in accordance with game $\mathrm{G}_0$. All in all the changes from game $\mathrm{G}_0$ to $\mathrm{G}_1$ are syntactical, and so $\Delta(\mathrm{G}_0, \mathrm{G}_1) = 0$.

*Note:* Since after game $\mathrm{G}_1$ the simulator no longer "queries" any external oracle, from now on we take the term *query* to exclusively mean "adversarial query": a query to either OnLeftQuery$(\cdot)$ or OnRightQuery$(\cdot)$, made by $\mathcal{A}$.

**($\mathbf{G_1} \rightarrow \mathbf{G_2}$; Figures 19 & 20.)** Game $\mathrm{G}_2$ adds a number of "bells and whistles" that have no effect except to result in many unused calls to Random(). Moreover, $\mathrm{G}_2$ maintains a set $X$ containing all values queried by the adversary, as well as all values sampled in Random(). The number of calls to Random() per query is normalized via the variable `NumCallsToRandom`, which forces Random() to be called exactly twice for every left oracle query and $4q_1 + 3$ times for every right oracle query. (Indeed, it is possible to check for $\mathrm{G}_2$—as well as for all other games—that Random() is called at most twice per left oracle query and at most $4q_1 + 3$ times per right oracle query. The latter maximum is achieved in game $\mathrm{G}_{11}$.) Besides wasting a portion of the random tape these changes have no effect, and so $\Delta(\mathrm{G}_1, \mathrm{G}_2) = 0$.

*Note:* For the remaining games, it will be convenient to view the randomness used in BuildXprime() as coming from a second, independent random tape; that is, we dedicate a "primary" random tape for the subroutine Random(), and use a "secondary" random tape for sampling done by BuildXprime(). For every right oracle query there is an associated sequence of $4q_1 + 3$ blocks used for that call on the primary random tape; we call such a sequence a *chunk* (of the primary random tape). We note that chunks are not necessarily contiguous on the primary random tape, given the presence of left oracle queries and that, moreover, there placement only becomes known at runtime, as the adversary makes queries.

**($G_2 \to G_3$; Figure 20.)** Since games $G_2$ and $G_3$ are identical up to $\mathsf{bad} \leftarrow \mathsf{true}$, it suffices to upper bound the probability of the latter event. Since $X \cup X'$ has size $(4q_1 + 3)q_2 + 2q_1$, and since Random() is called $4q_1 + 3$ times per query to OnRightQuery() which is itself called (at most) $q_2$ times by the adversary, the chance $\mathsf{bad} \leftarrow \mathsf{true}$ is triggered by one of the calls to OnRightQuery() is at most $((4q_1 + 3)q_2 + 2q_1)(4q_1 + 3)q_2/N$. Likewise, the chance of triggering $\mathsf{bad} \leftarrow \mathsf{true}$ by a call to OnLeftQuery() is at most $((4q_1 + 3)q_2 + 2q_1)2q_1/N$, since at most $q_1$ such calls are made, each giving rise to 2 calls to Random(). Adding these two bounds, we the chance of $\mathsf{bad} \leftarrow \mathsf{true}$ is at most $((4q_1 + 3)q_2 + 2q_1)^2/N$. Thus $\Delta(G_2, G_3) \leq ((4q_1 + 3)q_2 + 2q_1)^2/N$.

**($G_3 \to G_4$; Figures 20 & 21.)** Game $G_4$ introduces a set $Y$ that keeps track of values queried by the adversary and returned to the adversary. (These are, essentially, all the values "known to the adversary".) This and other minor changes (such as the insertion of a new $\mathsf{bad}$ flag) do not affect the execution, and thus $\Delta(G_3, G_4) = 0$.

($G_4 \to G_5$; **Figure 21.)** We argue that the probability of $\mathsf{bad} \leftarrow \mathsf{true}$ in game $G_5$ is upper bounded by $((4q_1 + 3)q_2 + 2q_1)(q_1 + q_2)/(N - 2q_2 - 2q_1)$. For this, we will show that when $\mathcal{A}$ makes its $(i + 1)$-th query $x_{i+1}$ to either of its oracles in game either $G_4$ or $G_5$ (but assuming the game has not aborted yet), it has no knowledge of the values in the set $X \backslash Y$, in the sense that, conditioned on its query history $\mathbf{Q}_i = \big((x_j, y_j, *_j)\big)_{j=1}^{i}$ so far, the set $X \backslash Y$ is equidistributed over all sets of size $(4q_1 + 3)q_{q_2} + 2q_{q_1} - |Y|$ in the complement of $Y$, where $q_{q_1}$ is the number of left oracle queries and $q_{q_2}$ is the number of right oracle queries among the adversary's first $i$ queries, with randomness computed with respect to all possible random tapes compatible with $\mathbf{Q}_i$. To argue this, let $X_{r,s}$ denote the value of the set $X$ when game $G_5$ is run with primary and secondary random tapes $r, s$ on the query sequence $\mathbf{Q}_i^*$, assuming that $r$ and $s$ are compatible with $\mathbf{Q}_i$. Fix such tapes $r, s$ (thus fixing $X_{r,s}$), and let $T$ be an arbitrary subset of $\{0,1\}^n \backslash Y$ of size $(4q_1 + 3)q_{q_2} + 2q_{q_1} - |Y|$ (where $Y = \{x_1, \ldots, y_i\}$). Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation taking $X_{r,s} \backslash Y$ to $T$ and fixing points in $Y$. Letting $\pi(r)$, $\pi(s)$ denote the application blockwise of $\pi$ to the random tapes $r$, $s$, it is easy to check that $X_{\pi(r),\pi(s)} = T \cup Y$, i.e. $X_{\pi(r),\pi(s)} \backslash Y = T$. It follows that $X \backslash Y$ is equidistributed among all sets of size $(4q_1+3)q_{q_2}+2q_{q_1}-|Y|$ in $\{0,1\}^n \backslash Y$, conditioned on $\mathbf{Q}_i$. Thus, the probability that $\mathsf{bad} \leftarrow \mathsf{true}$ at the $i$-th query is at most $((4q_1 + 3)q_{q_2} + 2q_{q_1} - |Y|)/(N - |Y|) \leq ((4q_1 + 3)q_2) + 2q_1)/(N - 2q_2 - 2q_1)$. Union bounding over all $q_1 + q_2$ queries made by the adversary gives $\Delta(G_4, G_5) \leq ((4q_1 + 3)q_2 + 2q_1)(q_1 + q_2)/(N - 2q_2 - 2q_1)$.

*Note:* If Abort does not occur in game $G_5$, then the graphs $\mathtt{G}$, $\mathtt{G}^{-1}$ and $\mathtt{g}$ can be shown to be fairly structured. For example, it is easy to check that after each query is answered (subject to non-abortion) the following three invariants are maintained in $G_5$:

(1) No cycles or collisions (every vertex has indegree[10] at most 1) occur in the directed graph defined by $\mathtt{G}$, or in the directed graphs defined by $\mathtt{G}^{-1}$ and $\mathtt{g}$;

(2) As the graph $\mathtt{G}$ grows, no edge is ever added whose endpoint is already adjacent to another edge of $\mathtt{G} \cup \mathtt{G}^{-1} \cup \mathtt{g}$;

(3) $\mathtt{G}^{-1}$ is a reversed subgraph of $\mathtt{G}$ (every edge of $\mathtt{G}^{-1}$, reversed, becomes an edge of $\mathtt{G}$).

To discuss further features of the graphs, we make some additional definitions. Let $\overline{\mathtt{G}}$ be the restriction of $\mathtt{G}$ to edges whose reversal is also in $\mathtt{G}$ (these are the edges "known to the simulator"). The edges in

[10]The outdegree is automatically at most 1, since these graphs represent functions.

$\mathtt{G}\backslash\overline{\mathtt{G}}$ are called *hidden.* A *ladder* is configuration consisting of an ordered pair of two maximal paths $(s_{-q_1},\ldots,s_0,\ldots,s_{m'})$ and $(t_{-k},\ldots,t_0,\ldots,t_m)$ in $\overline{\mathtt{G}}$, such that: (i) the paths are vertex disjoint; (ii) $0\leq k\leq q_1$, $m\geq q_1+1$ and $m'\in\{m-1,m\}$; (iii) $(t_i,s_i)$, $-k\leq i\leq m'$, and $(s_i,t_{i+1})$, $-k\leq i\leq m-1$, are edges of $\mathtt{g}$, and these are the only edges of $\mathtt{g}$ adjacent to a vertex in $\{s_{-q_1},\ldots,s_{m'},t_{-k},\ldots,t_m\}$.

The path $(s_{-q_1},\ldots,s_{m'})$ is called the ladder's *upper side* and the path $(t_{-k},\ldots,t_m)$ is called the ladder's *lower side*[11] The vertex $t_{-k}$ is called the *anchor* of a ladder, and $t_0$ is called the *center.* The edges $(t_i,s_i)$, $(s_i,t_{i+1})$, $-k\leq i<q_1$, are called the *rungs* of the ladder. A rung of the form $(t_i,s_i)$ is *upward*; a rung of the form $(s_i,t_{i+1})$ is *downward.* The *tip* of a ladder is the last vertex on the unique $\mathtt{g}$-path starting at the anchor. (The tip is either $t_m$ or $s_{m'}$ depending on whether $m>m'$ or $m=m'$.) The *tip precursor* is the next-to-last vertex on this path.

A maximal path in $\mathtt{G}\backslash\overline{\mathtt{G}}$ ending at a ladder's anchor is called an *anchored hidden chain.* A maximal path in $\mathtt{G}\backslash\overline{\mathtt{G}}$ not adjacent to any edges in $\overline{\mathtt{G}}\cup\mathtt{g}$ is called an *isolated hidden chain.* We make these definitions only for games $\mathrm{G}_5$ and $\mathrm{G}_6$ (afterwards, we will update the definitions as well as the stated properties). For games $\mathrm{G}_5$, $\mathrm{G}_6$ one can check that the following facts hold after each query is answered, presuming the game is in a non-aborted state:

(4) Any two ladders are vertex- (and hence also edge-) disjoint;

(5) Every edge of $\mathtt{G}\cup\mathtt{g}$ is either in a ladder, in a hidden chain anchored to a ladder, or in an isolated hidden chain; moreover (trivially) the endpoints of edges in $\mathtt{G}\backslash\overline{\mathtt{G}}$ are in the set $Y$;

(6) The set of vertices in a ladder that are in the set $Y$ forms a connected component in $\mathtt{G}\cup\mathtt{g}$ that contains the anchor of the ladder; moreover, this set does not contain any upper side vertices non-adjacent to a rung, and does not contain the ladder's tip.

Giving detailed proofs of (4), (5) and (6) for $\mathrm{G}_5$ and $\mathrm{G}_6$ is not difficult, but would take us afield. We note that the (high-level) reason why $Y$ never contains a ladder's tip is that the simulator keeps elongating each ladder such that every ladder's tip is at least distance $q_1+1$ in $\mathtt{G}\cup\mathtt{g}$ from any point in the ladder that $\mathcal{A}$ has queried to the simulator, coupled with the fact that $\mathcal{A}$ has only $q_1$ queries to the RO.

We make a few more definitions. The length $\ell$ of the longest path in $\mathtt{G}$ ending at the center $t_0$ of a ladder is called the *extension length* of the ladder; we note the path of length $\ell$ ending at $t_0$ is unique, and that its head is either the ladder's anchor or else the head of the (unique) hidden chain ending at the ladder's anchor. It is easy to check from (5), (6) and from the fact that the adversary makes at most $q_1$ queries that the extension length is at most $q_1$. Finally, if $(s_{-q_1},\ldots,s_{q_1})$ is the upper side of a ladder of extension length $\ell$, then we call the path $(s_{-q_1},\ldots,s_{-\ell})$ the *excess upper path.*

As hinted above, since successive games keep tweaking the data structures, an unavoidable annoyance is that some of the above definitions will become outdated (see, for example, the comment after the transition $\mathrm{G}_6\rightarrow\mathrm{G}_7$). We have found no other way, unfortunately, but to update each time those definitions that are still required for the discussion of subsequent games.

We also point out that games $\mathrm{G}_5$–$\mathrm{G}_{20}$ have the following well-behavedness property: *except for the abort event*, the execution path taken during one query answer cycle does not depend on the internal randomness; that is, knowing the state $\mathtt{G}^{\pm1}$ and $\mathtt{g}$ at the point when a query is asked is enough to know which lines of code will be executed by the game in what order until it returns, assuming Abort does not occur. This property will be useful for "randomness cut-and-paste" arguments used below.

[11] We note that since ladders are defined as an ordered pair of paths, there is no ambiguity as to which is the upper or lower side; nonetheless, even if a ladder is only given as the subgraph in $\mathtt{G}\cup\mathtt{g}$ induced by the path vertices, one can tell the upper from the lower side from the fact that the lower side contains the head of the $\mathtt{g}$-path formed by the ladder's rungs.

**($G_5 \to G_6$; Figures 21 & 22.)** The unique change in game $G_6$ is that the line "If $y = \perp$ then Abort" is removed from the procedure OnRightQuery($\cdot$). We argue that $\mathtt{G}^{-i}[\mathtt{g}[x_i]]$ at the previous line never returns $\perp$, so that this change has no effect. Indeed, given property (5) above, the edge $(x_i, \mathtt{g}[x_i])$ must be either an upward or a downward rung of some ladder. If it is a downward rung and $\mathtt{G}^{-i}[\mathtt{g}[x_i]] = \perp$ then $i > 0$ and $x = x_0$ must be an upper side vertex nonadjacent to any downward rung, a contradiction to the fact that $Y$ contains no such vertices and to the abortion condition added in game $G_5$. If $(x_i, \mathtt{g}[x_i])$ is an upward rung, then $\mathtt{G}^{-i}[\mathtt{g}[x_i]] \neq \perp$ follows from the fact that any ladder's extension length is at most $q_1$. Thus $\Delta(G_5, G_6) = 0$.

**($G_6 \to G_7$; Figures 22 & 23.)** In game $G_7$, the call SetTable($G, x$, ROsub) is moved from SimROsub() to ROsub(), and SimROsub() is bypassed altogether in favor of ROsub(). The effect of this change is that $\mathtt{G}^{-1}$ becomes the exact reverse graph of $\mathtt{G}$, instead of being a subgraph of the reversal of $\mathtt{G}$. However, this enlargement of $\mathtt{G}^{-1}$ has no effect, since no lookups in $\mathtt{G}^{-1}$ (or its iteration) in game $G_6$ ever give value $\perp$ (since the only use of $\mathtt{G}^{-1}$ occurs in OnRightQuery(), this was in fact argued above). Thus $\Delta(G_6, G_7) = 0$.

*Definition update.* Since $\mathtt{G}^{-1}$ is now the same as $\mathtt{G}$, $\overline{\mathtt{G}} = \mathtt{G}$ and some definitions must be adjusted. In particular, the lower side of a ladder $(t_{-k}, \dots, t_m)$ is not required to be a maximal path in $\mathtt{G}$, but only to be "non-extendible at the end" in $\mathtt{G}$ (i.e., that $t_m$ have outdegree 0 in $\mathtt{G}$). The definitions of upper side, anchor, center, tip and tip precursor, upward and downward rungs are unaffected. We redefine a anchored hidden chain to be a path in $\mathtt{G}$ ending at a ladder's anchor, that is "non-extendible at the start" (i.e., the path's head has indegree 0), and that is nonadjacent to any edges of $\mathtt{g}$ except at its endpoint (the ladder's anchor). Isolated hidden chains are redefined as (maximal) connected components of $\mathtt{G}$ not containing any vertex of a ladder or any vertex adjacent to an edge of $\mathtt{g}$. Subject to these changes, the above properties (1)–(6) still hold (though the second half of (5) is now void).

**($G_7 \to G_8$; Figures 23 & 24.)** In game $G_8$ MakeLadder() "manually" erases the excess upper path of the ladder being created. This has no effect because these entries of $\mathtt{G}$ and $\mathtt{G}^{-1}$ are never subsequently read. Thus $\Delta(G_7, G_8) = 0$.

*Definition update.* We allow the upper side of a ladder to be shortened. Specifically, the upper and lower sides of a ladder are now paths $(s_{-\ell}, \dots, s_{m'})$ and $(t_{-k}, \dots, t_m)$ in $\mathtt{G}$ such that $0 \leq k \leq \ell$, such that the upper side is a maximal path in $\mathtt{G}$ and $t_m$ has outdegree 0 in $\mathtt{G}$, such that properties (i), (ii), (iii) above for ladders still hold. Again, one can verify that items (1)–(6) still hold (from $G_8$ all the way to $G_{16}$, in fact). We also deprecate the definitions of "center" and "extension length" without updating them. (These will no longer be needed.)

**($G_8 \to G_9$; Figures 24 & 25.)** In game $G_9$ the excess upper path of the ladder is "never created in the first place". Note this results in fewer calls to Random() from within MakeLadder(). In particular, for example, the call to MakeLadder() has fewer chances of causing the game to abort. Nonetheless, we argue that games $G_8$ and $G_9$ are indistinguishable. Note that if one cuts the first $q_1 - \ell$ blocks of (primary tape) randomness used by MakeLadder during a given query and pastes these at the end the same chunk (for the $i$-th right oracle query, this is the $i$-th chunk), then, for this one query, and assuming given states of $\mathtt{G}, \mathtt{G}^{-1}$ and $\mathtt{g}$ at the start of the query, running $G_8$ with the original chunk or else running $G_9$ with the new chunk produces the exact same result, in the sense that the probability of abortion (computed only over the secondary random tape, and fixing the first) is equal in either world, and that if the game doesn't abort then the final values of the tables $\mathtt{G}$, $\mathtt{G}^{-1}$, $\mathtt{g}$ as well as of the set $X$ are equal in either world at the end of that query (in particular, the blocks that were cut-and-pasted become part of $X$ in either case). From this it is easy to conclude, by induction over the number of queries, that the two worlds are equidistributed. Thus $\Delta(\mathrm{G}_8, \mathrm{G}_9) = 0$.

**($\mathbf{G_9 \to G_{10}}$; Figures 25 & 26.)** The game $\mathrm{G}_{10}$ introduces a functionality whereby the value sampled in ROsub() can be set externally. This functionality is only used by MakeLadder(), which is honestly sampling the second argument to ROsub. Thus the change is syntactical, and $\Delta(\mathrm{G}_9, \mathrm{G}_{10}) = 0$.

**($\mathbf{G_{10} \to G_{11}}$; Figures 26 & 27.)** Game $\mathrm{G}_{11}$ introduces a new array `gProxy` that is sampled in Random(), and that is never used anywhere else. The sampling of `gProxy` in ROsub produces a "translation effect" on the random tape, but one can argue this effect leaves the games equidistributed by using a cut-and-paste argument as in the transition $\mathrm{G}_8 \to \mathrm{G}_9$. (We note in passing that the cut-and-paste argument is facilitated by the fact that we know at the outset of a right oracle query exactly which primary random blocks of the current chunk will be assigned to `gProxy`; see the last comment before the $\mathrm{G}_5 \to \mathrm{G}_6$ transition.) Thus we have $\Delta(\mathrm{G}_{10}, \mathrm{G}_{11}) = 0$.

**($\mathbf{G_{11} \to G_{12}}$; Figures 27 & 28.)** In game $\mathrm{G}_{12}$ the randomness "stored" in `gProxy` is accessed by MakeLadder() instead of using calls to Random(). This requires some justifications. Note that, since every value stored in `gProxy` is sampled by Random(), and hence is in $X$, then if one conditions only on knowledge of $\mathtt{G}$, $\mathtt{G}^{-1}$, $\mathtt{g}$ and on the fact that the currently queried value $x$ did not cause Abort (via collision with $X \backslash Y$), and one reads a "fresh" (yet-unread) value from `gProxy`, then this value is distributed randomly at uniform among all values that do not appear in $\mathtt{G}^{\pm 1}$, $\mathtt{g}^{\pm 1}$, and that are not equal to $x$—moreover, note the same statement holds if one replaces "reading a fresh value in `gProxy`" by "calling Random(), assuming Abort does not occur". Moreover, the fresh value read from `gProxy` is also stored in $X$, just like a value returned by Random(). The two methods of obtaining a random value therefore only differ in that Random() can cause Abort, whereas reading a fresh value from `gProxy` cannot. So *conditioned on non-abortion* and on a given state of $\mathtt{G}^{\pm 1}$ and $\mathtt{g}$ at the point when a query is issued, the answer to the query will be equidistributed in games $\mathrm{G}_{11}$ and $\mathrm{G}_{12}$; moreover, the state of $\mathtt{G}^{\pm 1}$ and $\mathtt{g}$ will also be equidistributed (allowing to pursue induction on the number of queries). Thus, since Abort anyway occurs with equal probability at each query of each game, $\Delta(\mathrm{G}_{11}, \mathrm{G}_{12}) = 0$.

*Note:* Game $\mathrm{G}_{12}$ already samples and uses all randomness "in the same order" as the final construction $G[g]$. One consequence of this is that games $\mathrm{G}_{12}$ through $\mathrm{G}_{20}$ are indistinguishable in the following strong sense: running these games with equal random tapes (primary and secondary) on the same query sequence gives the same exact query answers (or Abort answer). Nonetheless, this must be argued game by game.

**($\mathbf{G_{12} \to G_{13}}$; Figures 28 & 29.)** Game $\mathrm{G}_{13}$ removes the "freshness" and "definedness" checks on `gProxy` from $\mathrm{G}_{12}$. On the one hand, because a ladder is only created once and ladders are vertex-disjoint, all values $\mathtt{G}^{-\ell}[x]$ or $\mathtt{G}^{i+1}[x]$ used as indices in MakeLadder() are distinct, the same entry of `gProxy` cannot be read by two distinct calls to MakeLadder (or twice during the same call). On the other hand, note a fresh value $\mathtt{gProxy}[u]$ exists as long as a call of the form $\mathrm{ROsub}(u, \perp)$ has been made before $\mathtt{g}[u]$ is defined, and before any call of the form $\mathrm{ROsub}(u, y)$, $y \neq \perp$ has been made. For the case of $u = \mathtt{G}^{-i}[x]$, $i \geq 1$, such calls have been while answering the adversary's own queries to the left oracle. For the case of $u = \mathtt{G}^{i}[x]$, $i \geq 0$, such calls in the (necessarily completed) For loop of OnRightQuery() (and possibly before while answering $\mathcal{A}$'s left oracle queries). Hence the `gProxy` values read by MakeLadder() are always defined and fresh. Thus, $\Delta(\mathrm{G}_{12}, \mathrm{G}_{13}) = 0$.

**($\mathbf{G_{13} \to G_{14}}$; Figures 29 & 30.)** In $\mathrm{G}_{14}$ the line $\mathtt{gProxy}[\mathtt{gProxy}[x]] \leftarrow z$ is added to the If-block of ROsub(). Because we have argued above that MakeLadder() never reads a $\perp$ value from `gProxy`, it is sufficient to argue that $\mathtt{gProxy}[\mathtt{gProxy}[x]]$ is always $\perp$ right before $\mathtt{gProxy}[\mathtt{gProxy}[x]]$ is set in

ROsub() in order to argue that the change in ROsub() has no effect. (In particular, note that it is not our concern whether gProxy[gProxy[$x$]] is overwritten by the line gProxy[$x$] $\leftarrow$ g[$x$] during a later call to ROsub() with argument gProxy[$x$]; we are concerned with gProxy[gProxy[$x$]] being *read*, and influencing the execution, not with it being overwritten.) But since gProxy[$x$] has just been returned by Random(), and since ROsub() is always called with a first argument that is in $X$ (and since Random() samples outside of $X$), it is clear that gProxy[gProxy[$x$]] is $\perp$ right before being assigned $z$. Hence the change to ROsub() has no effect and $\Delta(\mathrm{G}_{13}, \mathrm{G}_{14}) = 0$.

**($\mathbf{G_{14} \to G_{15}}$; Figures 30 & 31.)** In game $\mathrm{G}_{15}$ the function ROsub() is rewritten, and takes (again) a single argument. We argue the changes have no effect. More precisely, we argue by induction on the number of calls to ROsub() that for the same query sequence $\mathbf{Q}^*_{q_2}$ and the same random coins (primary and secondary), the contents of the arrays G, g and gProxy are the exact same after each call to ROsub() in $\mathrm{G}_{14}$ and $\mathrm{G}_{15}$. (Obviously, thus, the sequence of query answers will also be the same.) Moreover we also argue by induction on the number of calls to ROsub() that, for both games $\mathrm{G}_{14}$ and $\mathrm{G}_{15}$, gProxy always contains g, in the sense that $\mathtt{g}[u] \neq \perp \implies \mathtt{gProxy}[u] = \mathtt{g}[u]$ for all $u \in \{0,1\}^n$, with this invariant being true, in fact, after each line of execution (internal or external to ROsub(), in either game). We note gProxy obviously contains g at the start of each game, when both arrays are empty.

Assuming the latter claim by induction, the If-block containing the instruction gProxy[$x$] $\leftarrow$ g[$x$] obviously has no effect in $\mathrm{G}_{14}$, so we may ignore it[12]. We note that at the point when MakeLadder() is called on a point $x_0$, and assuming $\ell$ as defined in MakeLadder(), there is a maximal path $(x_{-\ell}, \ldots, x_0, \ldots, x_{q_1+1})$ existing in G such that g is defined at none of the $x_i$'s, such that $s_i :=$ gProxy[$x_i$] is defined for $-\ell \leq i \leq q_1$, and such that gProxy[$s_i$] $= x_{i+1}$ also for $-\ell \leq i \leq q_1$; the ladder's upper side becomes $(s_{-\ell}, \ldots, s_{q_1})$ and gProxy is unchanged when MakeLadder() returns, so that gProxy, in particular, contains all the values of g set by FillInRungs() at the end of the call to MakeLadder(). Moreover, it is easy to check that future calls to FillInRungs() for the same ladder maintain the invariant that g is a subtable of gProxy, given that any call to ROsub($x$) or ROsub($x, \perp$) at a point $x$ such that g[$x$] is already defined results in setting gProxy[g[$x$]] = gProxy[gProxy[$x$]] to the call's value. Thus, g remains throughout a subtable of gProxy.

Note that if G[$x$] is defined then calls to ROsub() with first argument $x$ are obviously equivalent in $\mathrm{G}_{14}$ and $\mathrm{G}_{15}$. It therefore suffices to consider the cases when G[$x$] is undefined.

For calls ROsub($x, z$) with $z = \perp$ (and G[$x$] $= \perp$) in game $\mathrm{G}_{14}$, these are obviously equivalent to calling ROsub($x$) in $\mathrm{G}_{15}$ if gProxy[$x$] is undefined; if $y :=$ gProxy[$x$] is defined then the calls are still equivalent as long as gProxy[$y$] is undefined. However it is easy to check that the only calls for which G[$x$] $= \perp$ and gProxy[$x$], gProxy[$y$] are both defined are the calls made via MakeLadder(), for which $z \neq \perp$. This establishes that ROsub() has the same effect (on G, gProxy and g) in $\mathrm{G}_{14}$, $\mathrm{G}_{15}$ for all calls to ROsub() made outside MakeLadder(). For the calls to ROsub($x$)/ROsub($x, z$) made from within MakeLadder(), moreover, it is easy to see that gProxy[$x$], gProxy[$y$] = gProxy[gProxy[$x$]] are defined beforehand, and that the result of such a call is just to set G[$x$] $= z$ where $z =$ gProxy[$y$]. Hence, all calls to ROsub() are equivalent in either world, and $\Delta(\mathrm{G}_{14}, \mathrm{G}_{15}) = 0$.

**($\mathbf{G_{15} \to G_{16}}$; Figures 31 & 32.)** $\mathrm{G}_{16}$ replaces the array g by gProxy; the only difference between the two arrays in game $\mathrm{G}_{15}$ is that gProxy can be defined on more points than g (as established),

[12] An astute reader may note that this If-block is never used in the analysis of any transition, from the point in game $\mathrm{G}_{11}$ where it appears. In fact, this If-block is superfluous, strictly speaking, but is kept for esthetical reasons. More precisely, it is kept so that gProxy[$u$] and g[$u$] never contain different non-null values in games $\mathrm{G}_{11}$–$\mathrm{G}_{13}$; without the If-block such inconsistencies could (and would) occur when a ladder is "extended" (via calls to FillInRungs() made from within OnRightQuery()), but without effect, because not affecting entries of gProxy subsequently read by MakeLadder(). In $\mathrm{G}_{14}$, the If-block has no effect at all.

and this matters when $\mathsf{g}[x_i]$ is tested for a value in OnRightQuery(). Game $\mathrm{G}_{16}$ circumvents this by introducing boolean field $\mathsf{KnownToSim}$, which is set to $\mathsf{true}$ if and only if the corresponding entry of the "old" table $\mathsf{g}$ (in game $\mathrm{G}_{15}$) is set to a non-$\perp$ value. Moreover $\mathrm{G}_{16}$ introduces two new calls to SetTable() in ROsub(), but since the array $\mathsf{g}^{-1}$ is not used these have no effect. Altogether, therefore, $\Delta(\mathrm{G}_{15}, \mathrm{G}_{16}) = 0$.

**($\mathbf{G_{16} \to G_{17}}$; Figures 32 & 33.)** In game $\mathrm{G}_{17}$, $\mathtt{G}$ is replaced everywhere by $\mathsf{g}^2$ and $\mathtt{G}^{-1}$ is replaced everywhere by $\mathsf{g}^{-2}$. Since it is straightforward to check that, in game $\mathrm{G}_{16}$, $\mathtt{G}[x] = \mathsf{g}^2[x]$ for all $x$ where $\mathtt{G}[x] \neq \perp$, and also $\mathtt{G}^{-1}[x] = \mathsf{g}^{-2}[x]$ for all $x$ where $\mathtt{G}^{-1}[x] \neq \perp$, this change has no effect. Thus $\Delta(\mathrm{G}_{16}, \mathrm{G}_{17}) = 0$.

**($\mathbf{G_{17} \to G_{18}}$; Figures 33 & 34.)** Game $\mathrm{G}_{18}$ replaces the call to MakeLadder($x$) with simply FillInRungs($x$). Indeed it is easy to check that in game $\mathrm{G}_{17}$ the only part of MakeLadder() with any effect is the call to FillInRungs(). Moreover calls to SetTable($\mathsf{g}, *, *$) are dropped in $\mathrm{G}_{18}$ since $\mathsf{g}^{-1}$ is no longer used there. Thus the changes have no effect and $\Delta(\mathrm{G}_{17}, \mathrm{G}_{18}) = 0$.

**($\mathbf{G_{18} \to G_{19}}$; Figures 34 & 35.)** In game $\mathrm{G}_{19}$ calls to FillInRungs() have been "folded back" into the For loop of OnRightQuery(), to equivalent effect. Indeed it is just a matter of case checking to see that that $\mathrm{G}_{19}$ produces the same result as $\mathrm{G}_{18}$, whether FillInRungs() is called inside or outside of the For loop in $\mathrm{G}_{18}$. Thus $\Delta(\mathrm{G}_{18}, \mathrm{G}_{19}) = 0$.

**($\mathbf{G_{19} \to G_{20}}$; Figure 35.)** In game $\mathrm{G}_{20}$ all Abort conditions are dropped. Arguing as in transitions $\mathrm{G}_2 \to \mathrm{G}_3$ and $\mathrm{G}_4 \to \mathrm{G}_5$ (and using a union bound), one can show that probability of $\mathsf{bad} \leftarrow \mathsf{true}$ in $\mathrm{G}_{19}$ is at most $\Delta(\mathrm{G}_2, \mathrm{G}_3) + \Delta(\mathrm{G}_4, \mathrm{G}_5)$. Thus $\Delta(\mathrm{G}_{19}, \mathrm{G}_{20}) \leq ((4q_1 + 3)q_2 + 2q_1)^2/N + ((4q_1 + 3)q_2 + 2q_1)(q_1 + q_2)/(N - 2q_2 - 2q_1)$.

**($\mathbf{G_{20} \to G_{21}}$; Figures 35 & 36.)** Game $\mathrm{G}_{21}$ clears up clutter left over by the defunct Abort conditions, and removes all references to $\mathsf{KnownToSim}$, which was unused (because never tested for) in games $\mathrm{G}_{19}$ and $\mathrm{G}_{20}$, and therefore superfluous. The changes have no import and $\Delta(\mathrm{G}_{20}, \mathrm{G}_{21}) = 0$.

**($\mathbf{G_{21} \to G_{22}}$; Figure 36.)** Note that $\mathrm{G}_{21}$ always sets each (non-defined) entry of $\mathsf{g}$ to a lazy-sampled value, never overwrites entries of $\mathsf{g}$, and always returns $\mathsf{g}[x]$ to a query OnRightQuery($x$) and $\mathsf{g}[\mathsf{g}[x]]$ to a query OnLeftQuery($x$), like game $\mathrm{G}_{22}$. Thus, the equivalence of games $\mathrm{G}_{21}$ can be seen by using *early sampling* for $\mathsf{g}$, instead of lazy sampling, in which case both games are obviously equivalent. Thus $\Delta(\mathrm{G}_{21}, \mathrm{G}_{22}) = 0$.

Finally, summing the $\Delta$-values of transitions $\mathrm{G}_0 \to \mathrm{G}_1$ through $\mathrm{G}_{21} \to \mathrm{G}_{22}$, we find $\Delta(\mathrm{G}_2, \mathrm{G}_3) + \Delta(G_4, \mathrm{G}_5) + \Delta(\mathrm{G}_{19}, \mathrm{G}_{20}) = 2\Delta(\mathrm{G}_2, \mathrm{G}_3) + 2\Delta(G_4, \mathrm{G}_5)$, which is the bound advertised by the theorem.

**procedure** OnLeftQuery($x$): $\quad$ $G_0$

Ret RO($x$)

**procedure** OnRightQuery($x$):

$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\mathtt{g}[x_i] \neq \perp$ then
        $y \leftarrow \mathtt{G}^{-i}[\mathtt{g}[x_i]]$
        If $y = \perp$ then Abort
        FillInRungs($x, y$)
        Ret $y$
    SetTable($\mathtt{G}, x_i, \mathrm{RO}(x_i)$)
    $x_{i+1} \leftarrow \mathtt{G}[x_i]$
MakeLadder($x$)
Ret $\mathtt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\mathtt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    SetTable($\mathtt{G}, x_{i-1}, \mathrm{RO}(x_{i-1})$)
    $x_{i+1} \leftarrow \mathtt{G}[x_{i-1}]$
    SetTable($\mathtt{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$)

$s_{-q_1} \leftarrow_\$ \{0,1\}^n$
For $i = -q_1$ to $q_1 - 1$
    SetTable($\mathtt{G}, s_i, \mathrm{RO}(s_i)$)
    $s_{i+1} \leftarrow \mathtt{G}[s_i]$
FillInRungs($x, s_0$)

**subroutine** SetTable($\mathtt{T}, x, y$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret
$\mathtt{T}[x] = y$
$\mathtt{T}^{-1}[y] = x$

**procedure** OnLeftQuery($x$): $\quad$ $G_1$

Ret ROsub($x$)

**procedure** OnRightQuery($x$):

$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\mathtt{g}[x_i] \neq \perp$ then
        $y \leftarrow \mathtt{G}^{-i}[\mathtt{g}[x_i]]$
        If $y = \perp$ then Abort
        FillInRungs($x, y$)
        Ret $y$
    $x_{i+1} \leftarrow$ SimROsub($x_i$)
MakeLadder($x$)
Ret $\mathtt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\mathtt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ SimROsub($x_{i-1}$)
    SetTable($\mathtt{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$)

$s_{-q_1} \leftarrow$ Random()
For $i = -q_1$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ SimROsub($s_i$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x$)

If $\mathtt{G}[x]$ Ret $\mathtt{G}[x]$
Ret $\mathtt{G}[x] \leftarrow$ Random()

**subroutine** SimROsub($x$)

SetTable($\mathtt{G}, x, \mathrm{ROsub}(x)$)
Ret $\mathtt{G}[x]$

**subroutine** Random()

Ret $\leftarrow_\$ \{0,1\}^n$

**subroutine** SetTable($\mathtt{T}, x, y$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret
$\mathtt{T}[x] = y$
$\mathtt{T}^{-1}[y] = x$

Figure 19: The first two games used in the proof of Theorem 3.3. The adversary's left and right oracles are implemented by the procedures OnLeftQuery($\cdot$) and OnRightQuery($\cdot$). Game G0 is the "Ideal World", where the left oracle is implemented by a random oracle RO : $\{0,1\}^n \to \{0,1\}^n$ and the right oracle is the simulator $\mathcal{S}$, whose goal is to mimic a function $g : \{0,1\}^n \to \{0,1\}^n$ such that $\mathrm{RO} = g^2$.

```
procedure OnLeftQuery(x):                         G_2, [G_3]
NumCallsToRandom ← 0; MAX_CALLS ← 2
X ← X ∪ {x}
Finalization()
Ret ROsub(x)

procedure OnRightQuery(x):
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ then
        y ← G^{-i}[g[x_i]]
        If y = ⊥ then Abort
        FillInRungs(x, y)
        Break // (For loop)
    x_{i+1} ← SimROsub(x_i)
If i > q_1
    MakeLadder(x)
Finalization()
Ret g[x]

subroutine FillInRungs(x, y)
x_0 ← x, x_1 ← y
SetTable(g, x_0, x_1)
For i = 1 to 2q_1 + 1
    x_{i+1} ← SimROsub(x_{i-1})
    SetTable(g, x_i, x_{i+1})

subroutine MakeLadder(x)
s_{-q_1} ← Random()
For i = -q_1 to q_1 - 1
    s_{i+1} ← SimROsub(s_i)
FillInRungs(x, s_0)
```

```
subroutine ROsub(x)                               G_2, [G_3] (cont.)
If G[x] Ret G[x]
Ret G[x] ← Random()

subroutine SimROsub(x)
SetTable(G, x, ROsub(x))
Ret G[x]

subroutine Random()
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X'
    bad ← true
    [Abort]
X ← X ∪ {y}
NumCallsToRandom++
Ret y

subroutine SetTable(T, x, y)
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{-1}[y] = x

subroutine BuildXprime()
X' ← ∅
While |X| + |X'| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n \ (X ∪ X')
    X' ← X' ∪ {z}

subroutine Finalization()
While NumCallsToRandom < MAX_CALLS do
    Random()
```

Figure 20: Games $G_2$ and $G_3$ for the proof of Theorem 3.3. Game $G_3$ includes the boxed statement, game $G_2$ does not.

**procedure** OnLeftQuery($x$): $\quad$ $G_4$, $\boxed{G_5}$

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then
    $\textsf{bad} \leftarrow \textsf{true}$
    $\boxed{\text{Abort}}$
$X \leftarrow X \cup \{x\}$
ROsub($x$)
$Y \leftarrow Y \cup \{x, \texttt{G}[x]\}$
Finalization()
Ret $\texttt{G}[x]$

**procedure** OnRightQuery($x$):

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then
    $\textsf{bad} \leftarrow \textsf{true}$
    $\boxed{\text{Abort}}$
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\textsf{g}[x_i] \neq \perp$ then
        $y \leftarrow \texttt{G}^{-i}[\textsf{g}[x_i]]$
        If $y = \perp$ then Abort
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ SimROsub($x_i$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \textsf{g}[x]\}$
Finalization()
Ret $\textsf{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\textsf{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ SimROsub($x_{i-1}$)
    SetTable($\textsf{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$) $\quad$ $G_4$, $\boxed{G_5}$

$s_{-q_1} \leftarrow$ Random()
For $i = -q_1$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ SimROsub($s_i$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x$)

If $\texttt{G}[x]$ Ret $\texttt{G}[x]$
Ret $\texttt{G}[x] \leftarrow$ Random()

**subroutine** SimROsub($x$)

SetTable($\texttt{G}, x$, ROsub($x$))
Ret $\texttt{G}[x]$

**subroutine** Random()

$y \leftarrow_\$ \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\texttt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\texttt{T}, x, y$)

If $\texttt{T}[x]$ and $\texttt{T}[x] \neq y$ then Ret
$\texttt{T}[x] = y$
$\texttt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_\$ \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

**subroutine** Finalization()

While $\texttt{NumCallsToRandom} < \texttt{MAX\_CALLS}$ do
    Random()

Figure 21: Games $G_4$ and $G_5$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_6$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X′)\Y then Abort
X ← X ∪ {x}
ROsub(x)
Y ← Y ∪ {x, G[x]}
Finalization()
Ret G[x]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X′)\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ then
        y ← G^{−i}[g[x_i]]
        FillInRungs(x, y)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** FillInRungs($x, y$)

```
x_0 ← x, x_1 ← y
SetTable(g, x_0, x_1)
For i = 1 to 2q_1 + 1
    x_{i+1} ← SimROsub(x_{i−1})
    SetTable(g, x_i, x_{i+1})
```

**subroutine** MakeLadder($x$) $\mathrm{G}_6$ (cont.)

```
s_{−q_1} ← Random()
For i = −q_1 to q_1 − 1
    s_{i+1} ← ROsub(s_i)
FillInRungs(x, s_0)
```

**subroutine** ROsub($x$)

```
If G[x] Ret G[x]
Ret G[x] ← Random()
```

**subroutine** SimROsub($x$)

```
SetTable(G, x, ROsub(x))
Ret G[x]
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X′ then Abort
X ← X ∪ {y}
NumCallsToRandom++
Ret y
```

**subroutine** SetTable($\mathtt{T}, x, y$)

```
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{−1}[y] = x
```

**subroutine** BuildXprime()

```
X′ ← ∅
While |X| + |X′| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n\(X ∪ X′)
    X′ ← X′ ∪ {z}
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

Figure 22: Game $\mathrm{G}_6$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): G$_7$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X′)\Y then Abort
X ← X ∪ {x}
ROsub(x)
Y ← Y ∪ {x, G[x]}
Finalization()
Ret G[x]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X′)\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ then
        y ← G^{-i}[g[x_i]]
        FillInRungs(x, y)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** FillInRungs($x, y$)

```
x_0 ← x, x_1 ← y
SetTable(g, x_0, x_1)
For i = 1 to 2q_1 + 1
    x_{i+1} ← ROsub(x_{i-1})
    SetTable(g, x_i, x_{i+1})
```

**subroutine** MakeLadder($x$) G$_7$ (cont.)

```
s_{-q_1} ← Random()
For i = -q_1 to q_1 - 1
    s_{i+1} ← ROsub(s_i)
FillInRungs(x, s_0)
```

**subroutine** ROsub($x$)

```
If G[x] Ret G[x]
G[x] ← Random()
SetTable(G, x, G[x])
Ret G[x]
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X′ then Abort
X ← X ∪ {y}
NumCallsToRandom + +
Ret y
```

**subroutine** SetTable(T, $x, y$)

```
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{-1}[y] = x
```

**subroutine** BuildXprime()

```
X′ ← ∅
While |X| + |X′| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n\(X ∪ X′)
    X′ ← X′ ∪ {z}
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

Figure 23: Game G$_7$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_8$

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x$)
$Y \leftarrow Y \cup \{x, \texttt{G}[x]\}$
Finalization()
Ret $\texttt{G}[x]$

**procedure** OnRightQuery($x$):

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\texttt{g}[x_i] \neq \perp$ then
        $y \leftarrow \texttt{G}^{-i}[\texttt{g}[x_i]]$
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \texttt{g}[x]\}$
Finalization()
Ret $\texttt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\texttt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    SetTable($\texttt{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$) $\mathrm{G}_8$ (cont.)

$s_{-q_1} \leftarrow$ Random()
For $i = -q_1$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ ROsub($s_i$)
$\ell \leftarrow 0$
While $\texttt{G}^{(-\ell)}[x] \neq \perp$
    $\ell \leftarrow \ell + 1$
For $i = -q_1$ to $-\ell$
    $\texttt{G}^{-1}[s_i] \leftarrow \perp$
    If $i > \ell$ then $\texttt{G}[s_i] \leftarrow \perp$
FillInRungs($x, s_0$)

**subroutine** ROsub($x$)

If $\texttt{G}[x]$ Ret $\texttt{G}[x]$
$\texttt{G}[x] \leftarrow$ Random()
SetTable($\texttt{G}, x, \texttt{G}[x]$)
Ret $\texttt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\texttt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\texttt{T}, x, y$)

If $\texttt{T}[x]$ and $\texttt{T}[x] \neq y$ then Ret
$\texttt{T}[x] = y$
$\texttt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

**subroutine** Finalization()

While $\texttt{NumCallsToRandom} < \texttt{MAX\_CALLS}$ do
    Random()

Figure 24: Game $\mathrm{G}_8$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\quad$ $G_9$

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x$)
$Y \leftarrow Y \cup \{x, \texttt{G}[x]\}$
Finalization()
Ret $\texttt{G}[x]$

**procedure** OnRightQuery($x$):

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\texttt{g}[x_i] \neq \perp$ then
        $y \leftarrow \texttt{G}^{-i}[\texttt{g}[x_i]]$
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \texttt{g}[x]\}$
Finalization()
Ret $\texttt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\texttt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    SetTable($\texttt{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$) $\quad$ $G_9$ (cont.)

$\ell \leftarrow 0$
While $\texttt{G}^{(-\ell)}[x] \neq \perp$
    $\ell \leftarrow \ell + 1$
$s_{-\ell} \leftarrow$ Random()
For $i = -\ell$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ ROsub($s_i$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x$)

If $\texttt{G}[x]$ Ret $\texttt{G}[x]$
$\texttt{G}[x] \leftarrow$ Random()
SetTable($\texttt{G}, x, \texttt{G}[x]$)
Ret $\texttt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\texttt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\texttt{T}, x, y$)

If $\texttt{T}[x]$ and $\texttt{T}[x] \neq y$ then Ret
$\texttt{T}[x] = y$
$\texttt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

**subroutine** Finalization()

While $\texttt{NumCallsToRandom} < \texttt{MAX\_CALLS}$ do
    Random()

Figure 25: Game $G_9$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\quad$ $\mathrm{G}_{10}$

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x, \perp$)
$Y \leftarrow Y \cup \{x, \mathtt{G}[x]\}$
Finalization()
Ret $\mathtt{G}[x]$

**procedure** OnRightQuery($x$):

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\mathtt{g}[x_i] \neq \perp$ then
        $y \leftarrow \mathtt{G}^{-i}[\mathtt{g}[x_i]]$
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i, \perp$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \mathtt{g}[x]\}$
Finalization()
Ret $\mathtt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\mathtt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    SetTable($\mathtt{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$) $\quad$ $\mathrm{G}_{10}$ (cont.)

$\ell \leftarrow 0$
While $\mathtt{G}^{(-\ell)}[x] \neq \perp$
    $\ell \leftarrow \ell + 1$
$s_{-\ell} \leftarrow$ Random()
For $i = -\ell$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ Random()
    ROsub($s_i, s_{i+1}$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x, z$)

If $\mathtt{G}[x]$ Ret $\mathtt{G}[x]$
If $z = \perp$ then $z \leftarrow$ Random()
SetTable($\mathtt{G}, x, z$)
Ret $\mathtt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\mathtt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\mathtt{T}, x, y$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret
$\mathtt{T}[x] = y$
$\mathtt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

**subroutine** Finalization()

While $\mathtt{NumCallsToRandom} < \mathtt{MAX\_CALLS}$ do
    Random()

Figure 26: Game $\mathrm{G}_{10}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_{11}$

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x, \perp$)
$Y \leftarrow Y \cup \{x, \mathtt{G}[x]\}$
Finalization()
Ret $\mathtt{G}[x]$

**procedure** OnRightQuery($x$):

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\mathsf{g}[x_i] \neq \perp$ then
        $y \leftarrow \mathtt{G}^{-i}[\mathsf{g}[x_i]]$
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i, \perp$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \mathsf{g}[x]\}$
Finalization()
Ret $\mathsf{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\mathsf{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    SetTable($\mathsf{g}, x_i, x_{i+1}$)

**subroutine** MakeLadder($x$) $\mathrm{G}_{11}$ (cont.)

$\ell \leftarrow 0$
While $\mathtt{G}^{(-\ell)}[x] \neq \perp$
    $\ell \leftarrow \ell + 1$
$s_{-\ell} \leftarrow$ Random()
For $i = -\ell$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ Random()
    ROsub($s_i, s_{i+1}$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x, z$)

If $\mathtt{G}[x]$ Ret $\mathtt{G}[x]$
If $z = \perp$
    If $\mathsf{g}[x]$
        $\mathtt{gProxy}[x] \leftarrow \mathsf{g}[x]$
    If $\mathtt{gProxy}[x] = \perp$
        $\mathtt{gProxy}[x] \leftarrow$ Random()
        $\mathtt{gProxy}[x].\mathsf{fresh} \leftarrow \mathsf{true}$
    $z \leftarrow$ Random()
SetTable($\mathtt{G}, x, z$)
Ret $\mathtt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\mathtt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\mathtt{T}, x, y$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret
$\mathtt{T}[x] = y$
$\mathtt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

**subroutine** Finalization()

While $\mathtt{NumCallsToRandom} < \mathtt{MAX\_CALLS}$ do
    Random()

Figure 27: Game $\mathrm{G}_{11}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_{12}$

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x, \perp$)
$Y \leftarrow Y \cup \{x, \mathtt{G}[x]\}$
Finalization()
Ret $\mathtt{G}[x]$

**procedure** OnRightQuery($x$):

$\mathtt{NumCallsToRandom} \leftarrow 0$; $\mathtt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
  If $\mathtt{g}[x_i] \neq \perp$ then
    $y \leftarrow \mathtt{G}^{-i}[\mathtt{g}[x_i]]$
    FillInRungs($x, y$)
    Break // (For loop)
  $x_{i+1} \leftarrow$ ROsub($x_i, \perp$)
If $i > q_1$
  MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \mathtt{g}[x]\}$
Finalization()
Ret $\mathtt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\mathtt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
  $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
  SetTable($\mathtt{g}, x_i, x_{i+1}$)

**subroutine** Finalization()

While $\mathtt{NumCallsToRandom} < \mathtt{MAX\_CALLS}$ do
  Random()

**subroutine** MakeLadder($x$) $\mathrm{G}_{12}$ (cont.)

$\ell \leftarrow 0$
While $\mathtt{G}^{(-\ell)}[x] \neq \perp$
  $\ell \leftarrow \ell + 1$
$s_{-\ell} \leftarrow$ GetFresh($\mathtt{gProxy}, \mathtt{G}^{-\ell}[x]$)
For $i = -\ell$ to $q_1 - 1$
  $s_{i+1} \leftarrow$ GetFresh($\mathtt{gProxy}, \mathtt{G}^{i+1}[x]$)
  ROsub($s_i, s_{i+1}$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x, z$)

If $\mathtt{G}[x]$ Ret $\mathtt{G}[x]$
If $z = \perp$
  If $\mathtt{g}[x]$
    $\mathtt{gProxy}[x] \leftarrow \mathtt{g}[x]$
  If $\mathtt{gProxy}[x] = \perp$
    $\mathtt{gProxy}[x] \leftarrow$ Random()
    $\mathtt{gProxy}[x].\mathsf{fresh} \leftarrow \mathsf{true}$
  $z \leftarrow$ Random()
SetTable($\mathtt{G}, x, z$)
Ret $\mathtt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\mathtt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\mathtt{T}, x, y$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x] \neq y$ then Ret
$\mathtt{T}[x] = y$
$\mathtt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
  $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
  $X' \leftarrow X' \cup \{z\}$

**subroutine** GetFresh($\mathtt{T}, x$)

If $\mathtt{T}[x]$ and $\mathtt{T}[x].\mathsf{fresh} = \mathsf{true}$
  $\mathtt{T}[x].\mathsf{fresh} = \mathsf{false}$
  Ret $\mathtt{T}[x]$
Ret Random()

Figure 28: Game $\mathrm{G}_{12}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\quad \mathrm{G}_{13}$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
ROsub(x, ⊥)
Y ← Y ∪ {x, G[x]}
Finalization()
Ret G[x]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ then
        y ← G^{-i}[g[x_i]]
        FillInRungs(x, y)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i, ⊥)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** FillInRungs($x, y$)

```
x_0 ← x, x_1 ← y
SetTable(g, x_0, x_1)
For i = 1 to 2q_1 + 1
    x_{i+1} ← ROsub(x_{i-1})
    SetTable(g, x_i, x_{i+1})
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

**subroutine** MakeLadder($x$) $\quad \mathrm{G}_{13}$ (cont.)

```
ℓ ← 0
While G^{(-ℓ)}[x] ≠ ⊥
    ℓ ← ℓ + 1
s_{-ℓ} ← gProxy[G^{-ℓ}[x]]
For i = -ℓ to q_1 - 1
    s_{i+1} ← gProxy[G^{i+1}[x]]
    ROsub(s_i, s_{i+1})
FillInRungs(x, s_0)
```

**subroutine** ROsub($x, z$)

```
If G[x] Ret G[x]
If z = ⊥
    If g[x]
        gProxy[x] ← g[x]
    If gProxy[x] = ⊥
        gProxy[x] ← Random()
    z ← Random()
SetTable(G, x, z)
Ret G[x]
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X' then Abort
X ← X ∪ {y}
NumCallsToRandom++
Ret y
```

**subroutine** SetTable(T, $x, y$)

```
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{-1}[y] = x
```

**subroutine** BuildXprime()

```
X' ← ∅
While |X| + |X'| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n \ (X ∪ X')
    X' ← X' ∪ {z}
```

Figure 29: Game $\mathrm{G}_{13}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\quad$ $G_{14}$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
ROsub(x, ⊥)
Y ← Y ∪ {x, G[x]}
Finalization()
Ret G[x]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ then
        y ← G^{-i}[g[x_i]]
        FillInRungs(x, y)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i, ⊥)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** FillInRungs($x, y$)

```
x_0 ← x, x_1 ← y
SetTable(g, x_0, x_1)
For i = 1 to 2q_1 + 1
    x_{i+1} ← ROsub(x_{i-1})
    SetTable(g, x_i, x_{i+1})
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

**subroutine** MakeLadder($x$) $\quad$ $G_{14}$ (cont.)

```
ℓ ← 0
While G^{(-ℓ)}[x] ≠ ⊥
    ℓ ← ℓ + 1
s_{-ℓ} ← gProxy[G^{-ℓ}[x]]
For i = -ℓ to q_1 - 1
    s_{i+1} ← gProxy[G^{i+1}[x]]
    ROsub(s_i, s_{i+1})
FillInRungs(x, s_0)
```

**subroutine** ROsub($x, z$)

```
If G[x] Ret G[x]
If z = ⊥
    If g[x]
        gProxy[x] ← g[x]
    If gProxy[x] = ⊥
        gProxy[x] ← Random()
    z ← Random()
    gProxy[gProxy[x]] ← z
SetTable(G, x, z)
Ret G[x]
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X' then Abort
X ← X ∪ {y}
NumCallsToRandom + +
Ret y
```

**subroutine** SetTable(T, $x, y$)

```
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{-1}[y] = x
```

**subroutine** BuildXprime()

```
X' ← ∅
While |X| + |X'| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n\(X ∪ X')
    X' ← X' ∪ {z}
```

Figure 30: Game $G_{14}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_{15}$

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x$)
$Y \leftarrow Y \cup \{x, \texttt{G}[x]\}$
Finalization()
Ret $\texttt{G}[x]$

**procedure** OnRightQuery($x$):

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\texttt{g}[x_i] \neq \perp$ then
        $y \leftarrow \texttt{G}^{-i}[\texttt{g}[x_i]]$
        FillInRungs($x, y$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i$)
If $i > q_1$
    MakeLadder($x$)
$Y \leftarrow Y \cup \{x, \texttt{g}[x]\}$
Finalization()
Ret $\texttt{g}[x]$

**subroutine** FillInRungs($x, y$)

$x_0 \leftarrow x, x_1 \leftarrow y$
SetTable($\texttt{g}, x_0, x_1$)
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    SetTable($\texttt{g}, x_i, x_{i+1}$)

**subroutine** Finalization()

While $\texttt{NumCallsToRandom} < \texttt{MAX\_CALLS}$ do
    Random()

**subroutine** MakeLadder($x$) $\mathrm{G}_{15}$ (cont.)

$\ell \leftarrow 0$
While $\texttt{G}^{(-\ell)}[x] \neq \perp$
    $\ell \leftarrow \ell + 1$
$s_{-\ell} \leftarrow \texttt{gProxy}[\texttt{G}^{-\ell}[x]]$
For $i = -\ell$ to $q_1 - 1$
    $s_{i+1} \leftarrow$ ROsub($s_i$)
FillInRungs($x, s_0$)

**subroutine** ROsub($x$)

If $\texttt{G}[x]$ Ret $\texttt{G}[x]$
If $\texttt{gProxy}[x]$
    $y \leftarrow \texttt{gProxy}[x]$
else
    $y \leftarrow \texttt{gProxy}[x] \leftarrow$ Random()
If $\texttt{gProxy}[y]$
    $z \leftarrow \texttt{gProxy}[y]$
else
    $z \leftarrow \texttt{gProxy}[y] \leftarrow$ Random()
SetTable($\texttt{G}, x, z$)
Ret $\texttt{G}[x]$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\texttt{NumCallsToRandom}++$
Ret $y$

**subroutine** SetTable($\texttt{T}, x, y$)

If $\texttt{T}[x]$ and $\texttt{T}[x] \neq y$ then Ret
$\texttt{T}[x] = y$
$\texttt{T}^{-1}[y] = x$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

Figure 31: Game $\mathrm{G}_{15}$ for the proof of Theorem 3.3.

```
procedure OnLeftQuery(x):                                G_16
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
ROsub(x)
Y ← Y ∪ {x, G[x]}
Finalization()
Ret G[x]

procedure OnRightQuery(x):
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ and g[x_i].KnownToSim then
        FillInRungs(x)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]

subroutine FillInRungs(x)
x_0 ← x
x_1 ← g[x_0]
g[x_0].KnownToSim ← true
For i = 1 to 2q_1 + 1
    x_{i+1} ← ROsub(x_{i-1})
    g[x_i].KnownToSim ← true

subroutine Finalization()
While NumCallsToRandom < MAX_CALLS do
    Random()
```

```
subroutine MakeLadder(x)                                 G_16 (cont.)
ℓ ← 0
While G^(−ℓ)[x] ≠ ⊥
    ℓ ← ℓ + 1
s_{−ℓ} ← g[G^{−ℓ}[x]]
For i = −ℓ to q_1 − 1
    s_{i+1} ← ROsub(s_i)
FillInRungs(x)

subroutine ROsub(x)
If G[x] Ret G[x]
If g[x]
    y ← g[x]
else
    y ← Random()
    SetTable(g, x, y)
If g[y]
    z ← g[y]
else
    z ← Random()
    SetTable(g, y, z)
SetTable(G, x, z)
Ret G[x]

subroutine Random()
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X' then Abort
X ← X ∪ {y}
NumCallsToRandom ++
Ret y

subroutine SetTable(T, x, y)
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{−1}[y] = x

subroutine BuildXprime()
X' ← ∅
While |X| + |X'| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n\(X ∪ X')
    X' ← X' ∪ {z}
```

Figure 32: Game $G_{16}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_{17}$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
ROsub(x)
Y ← Y ∪ {x, g[g[x]]}
Finalization()
Ret g[g[x]]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X')\Y then Abort
X ← X ∪ {x}
x_0 ← x
For i = 0 to q_1
    If g[x_i] ≠ ⊥ and g[x_i].KnownToSim then
        FillInRungs(x)
        Break // (For loop)
    x_{i+1} ← ROsub(x_i)
If i > q_1
    MakeLadder(x)
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** FillInRungs($x$)

```
x_0 ← x
x_1 ← g[x_0]
g[x_0].KnownToSim ← true
For i = 1 to 2q_1 + 1
    x_{i+1} ← ROsub(x_{i-1})
    g[x_i].KnownToSim ← true
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

**subroutine** MakeLadder($x$) $\mathrm{G}_{17}$ (cont.)

```
ℓ ← 0
While g^{(-2ℓ)}[x] ≠ ⊥
    ℓ ← ℓ + 1
s_{-ℓ} ← g[g^{-2ℓ}[x]]
For i = -ℓ to q_1 - 1
    s_{i+1} ← ROsub(s_i)
FillInRungs(x)
```

**subroutine** ROsub($x$)

```
If g[x]
    y ← g[x]
else
    y ← Random()
    SetTable(g, x, y)
If g[y]
    z ← g[y]
else
    z ← Random()
    SetTable(g, y, z)
Ret z
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X' then Abort
X ← X ∪ {y}
NumCallsToRandom + +
Ret y
```

**subroutine** SetTable(T, $x$, $y$)

```
If T[x] and T[x] ≠ y then Ret
T[x] = y
T^{-1}[y] = x
```

**subroutine** BuildXprime()

```
X' ← ∅
While |X| + |X'| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n \ (X ∪ X')
    X' ← X' ∪ {z}
```

Figure 33: Game $\mathrm{G}_{17}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\mathrm{G}_{18}$

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 2$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
ROsub($x$)
$Y \leftarrow Y \cup \{x, \mathsf{g}[\mathsf{g}[x]]\}$
Finalization()
Ret $\mathsf{g}[\mathsf{g}[x]]$

**procedure** OnRightQuery($x$):

$\texttt{NumCallsToRandom} \leftarrow 0$; $\texttt{MAX\_CALLS} \leftarrow 4q_1+3$
BuildXprime()
If $x \in (X \cup X')\backslash Y$ then Abort
$X \leftarrow X \cup \{x\}$
$x_0 \leftarrow x$
For $i = 0$ to $q_1$
    If $\mathsf{g}[x_i] \neq \perp$ and $\mathsf{g}[x_i]$.KnownToSim then
        FillInRungs($x$)
        Break // (For loop)
    $x_{i+1} \leftarrow$ ROsub($x_i$)
If $i > q_1$
    FillInRungs($x$)
$Y \leftarrow Y \cup \{x, \mathsf{g}[x]\}$
Finalization()
Ret $\mathsf{g}[x]$

**subroutine** FillInRungs($x$)

$x_0 \leftarrow x$
$x_1 \leftarrow \mathsf{g}[x_0]$
$\mathsf{g}[x_0]$.KnownToSim $\leftarrow$ true
For $i = 1$ to $2q_1 + 1$
    $x_{i+1} \leftarrow$ ROsub($x_{i-1}$)
    $\mathsf{g}[x_i]$.KnownToSim $\leftarrow$ true

**subroutine** Finalization()

While $\texttt{NumCallsToRandom} < \texttt{MAX\_CALLS}$ do
    Random()

**subroutine** ROsub($x$) $\mathrm{G}_{18}$

If $\mathsf{g}[x]$
    $y \leftarrow \mathsf{g}[x]$
else
    $y \leftarrow \mathsf{g}[x] \leftarrow$ Random()
If $\mathsf{g}[y]$
    $z \leftarrow \mathsf{g}[y]$
else
    $z \leftarrow \mathsf{g}[y] \leftarrow$ Random()
Ret $z$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
BuildXprime()
If $y \in X \cup X'$ then Abort
$X \leftarrow X \cup \{y\}$
$\texttt{NumCallsToRandom}++$
Ret $y$

**subroutine** BuildXprime()

$X' \leftarrow \emptyset$
While $|X| + |X'| < (4q_1 + 3)q_2 + 2q_1$ do
    $z \leftarrow_{\$} \{0,1\}^n \backslash (X \cup X')$
    $X' \leftarrow X' \cup \{z\}$

Figure 34: Game $\mathrm{G}_{18}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\boxed{G_{19}}$ $G_{20}$

```
NumCallsToRandom ← 0; MAX_CALLS ← 2
BuildXprime()
If x ∈ (X ∪ X′)\Y
    bad ← true
    [Abort]
X ← X ∪ {x}
ROsub(x)
Y ← Y ∪ {x, g[g[x]]}
Finalization()
Ret g[g[x]]
```

**procedure** OnRightQuery($x$):

```
NumCallsToRandom ← 0; MAX_CALLS ← 4q_1+3
BuildXprime()
If x ∈ (X ∪ X′)\Y
    bad ← true
    [Abort]
X ← X ∪ {x}
x_0 ← x
For i = 1 to 2q_1 + 1
    ROsub(x_{i−1})
    x_i ← g[x_{i−1}]
    g[x_{i−1}].KnownToSim ← true
    g[x_i].KnownToSim ← true
Y ← Y ∪ {x, g[x]}
Finalization()
Ret g[x]
```

**subroutine** Finalization()

```
While NumCallsToRandom < MAX_CALLS do
    Random()
```

**subroutine** ROsub($x$) $\boxed{G_{19}}$ $G_{20}$

```
If g[x]
    y ← g[x]
else
    y ← g[x] ← Random()
If g[y]
    z ← g[y]
else
    z ← g[y] ← Random()
Ret z
```

**subroutine** Random()

```
y ←$ {0,1}^n
BuildXprime()
If y ∈ X ∪ X′
    bad ← true
    [Abort]
X ← X ∪ {y}
NumCallsToRandom ++
Ret y
```

**subroutine** BuildXprime()

```
X′ ← ∅
While |X| + |X′| < (4q_1 + 3)q_2 + 2q_1 do
    z ←$ {0,1}^n \ (X ∪ X′)
    X′ ← X′ ∪ {z}
```

Figure 35: Games $G_{19}$ and $G_{20}$ for the proof of Theorem 3.3.

**procedure** OnLeftQuery($x$): $\quad$ $G_{21}$

ROsub($x$)
Ret $\mathsf{g}[\mathsf{g}[x]]$

**procedure** OnRightQuery($x$):

$x_0 \leftarrow x$
For $i = 1$ to $2q_1 + 1$
    ROsub($x_{i-1}$)
    $x_i \leftarrow \mathsf{g}[x_{i-1}]$
Ret $\mathsf{g}[x]$

**subroutine** ROsub($x$)

If $\mathsf{g}[x] = \perp$
    $\mathsf{g}[x] \leftarrow$ Random()
$y \leftarrow \mathsf{g}[x]$
If $\mathsf{g}[y] = \perp$
    $\mathsf{g}[y] \leftarrow$ Random()
$z \leftarrow \mathsf{g}[y]$
Ret $z$

**subroutine** Random()

$y \leftarrow_{\$} \{0,1\}^n$
Ret $y$

**procedure** OnLeftQuery($x$): $\quad$ $G_{22}$

ROsub($x$)
Ret $\mathsf{g}[\mathsf{g}[x]]$

**procedure** OnRightQuery($x$):

If $\mathsf{g}[x] = \perp$
    $\mathsf{g}[x] \leftarrow_{\$} \{0,1\}^n$
Ret $\mathsf{g}[x]$

**subroutine** ROsub($x$)

If $\mathsf{g}[x] = \perp$
    $\mathsf{g}[x] \leftarrow_{\$} \{0,1\}^n$
$y \leftarrow \mathsf{g}[x]$
If $\mathsf{g}[y] = \perp$
    $\mathsf{g}[y] \leftarrow_{\$} \{0,1\}^n$
Ret $\mathsf{g}[y]$

Figure 36: Games $G_{21}$ and $G_{22}$ for the proof of Theorem 3.3.