On the Indifferentiability of the Integrated-Key Hash Functions

Saif Al-Kuwari

Department of Computer Science,
University of Bath, Bath, BA2 7AY, UK
S.Alkuwari@bath.ac.uk

Abstract

Most of today's popular hash functions are keyless such that they accept variable-length messages and return fixed-length fingerprints. However, recent separation results reported on several serious inherent weaknesses in these functions, motivating the design of hash functions in the keyed setting. The challenge in this case, however, is that on one hand, it is economically undesirable to abundant the already adopted (keyless) functions in favour of new (keyed) ones, and on the other hand, the process of converting a keyless function to a keyed one is, evidently, non-trivial. A solution to this dilemma is to adopt the "integrated-key" approach that creates keyed hash functions out of "unmodified" keyless primitives. In this paper, we adopt several integrated-key constructions and prove that they are indifferentiable from random oracle, showing in details how to develop indifferentiability proofs at the integrated-key setting. The presented indifferentiability proof is generic and can be applied on other hash functions constructed in this setting with sufficiently similar structures to the constructions in this paper.

1 Introduction

Cryptographic hash functions are the workhorses of cryptography. A classical problem in the hash function literature is how to formally argue about the security of a hash function without the presence of keys [16]. This problem, along with numerous cryptanalytic results [17, 18, 19] on some of the most popular keyless hash functions, clearly demonstrated that there are inherent weaknesses in the keyless design approach. A simple solution to these problems is to instead shift to *keyed* hash function, but adopting new hash functions neither economical nor easy to do. In most cases, it is much more convenient (and cheaper) to somehow patch an existing keyless hash function to adapt for a key rather than shifting to a new keyed one, but then the underlying building blocks of the keyless hash function will need to undertake non-trivial modifications since they do not naturally accommodate the key input. That is, a typical keyless hash function consists of two components, a keyless compression function $f : \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n$, and a construction $C^f : \{0,1\}^* \to \{0,1\}^n$ with access to f, which accepts a message $M \in \{0,1\}^*$ of variable-length, divides it into m-bit blocks (pads if necessary), and hashes the blocks iteratively by repeatedly calling f. On contrast, in a typical keyed setting, the compression function f is keyed admits an extra key input $f_K : \{0,1\}^k \times \{0,1\}^m \times \{0,1\}^n \to \{0,1\}^n$. Clearly, converting f to f_K, without major modifications to the underlying structure of f, is difficult, if at all possible. In [1] a moderate solution to this problem was proposed which seamlessly transforms keyless hash functions into keyed ones without "touching" the underlying keyless compression function, this new setting is called the *Integrated-Key* setting, which introduce a dedicated mixing function, called the integration function, to handle the key independently outside the compression function.

Integrated-Key Setting. A hash function constructed in the integrated-key setting $C^{f,g} : \mathcal{K} \times \mathcal{M} \to \{0,1\}^n$ is a family of hash functions utilizing a keyless compression function $f : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ and a keyed integration-function $g : \{0,1\}^k \times \{0,1\}^b \to \{0,1\}^b$ where the latter processes the key $K \in \mathcal{K}$ independently outside f. We require g to be called at least whenever f is called in a similar way as in the dedicated-key setting [3]. In general, keyed hash functions are less efficient than keyless ones because in the former an extra input (i.e. the key) needs to be processed. However, the efficiency of hash functions constructed in the integrated-key setting highly depends on the implementation of the corresponding integration function, which should ideally be a lightweight function. In section 2 we suggest adopting the prepend-permute-chop paradigm [10] for the integration function, which is both efficient and secure. We emphasize that it is not the focus of this paper to argue about the principles of the integrated-key setting (these were discussed in length in [1]), rather we are here mainly concerned about the indifferentiability of hash functions constructed in this setting.

Notation. The notation $x \xleftarrow{\$} \{0,1\}^n$ indicates that a value chosen uniformly at random from the set $\{0,1\}^n$ is assigned to the variable x. We denote by $|M|$ the length of the string M. Concatenation of n blocks is denoted by $M_1||\ldots||M_n$, or sometimes $M_1 \ldots M_n$. We refer to all values in the column labeled $\hat{x}$ of the table $\mathcal{T}_K$ by $\mathcal{T}_K(\hat{x})$. The empty string is denoted by $\perp$. We use the subscript $\_$ (underscore) to index an arbitrarily chosen row in a table. The notation X^Y indicates that a component X has an oracle access to another component Y (oracle access here means that Y is a public component that replies to any query it receives). We use calligraphic font to denote ideal components, such as a random oracle $\mathcal{F}$.

Organization. This paper is organized as follows, in section 2 we introduce the three integrated-key constructions from [1], followed by section 3 where we recall the indifferentiability framework

due to Coron *et al.* [8]. Section 4 is the main part (and contribution) of this paper where we provide a detailed indifferentiability proof of the constructions presented in section 2, nothing, however, that we developed the proof in a generic way which makes it applicable to other integrated-key constructions with sufficient similarities to the ones presented in this paper.

2 The iMD Constructions

Most of the popular hash functions (e.g. MD5, SHA-1) are based on the (keyless) Merkle-Damgård construction [13, 9]. Thus, in this work we adopt integrated-key variants of Merkle-Damgård [1] and prove that they are indifferentiable from $\mathcal{RO}$. Although both the compression and integration functions can be visualized as a single entity, to improve the accuracy of the indifferentiability bounds in the proof, we make an explicit distinction between the two and show how indifferentiability proofs are carried out in the integrated-key setting. Figure 1 defines the x-iMD, y-iMD and c-iMD constructions[1][1], where $Pad_s(.)$ is a suffix-free padding.

x-iMD$^{f_x,g_x}(K,M)$:	y-iMD$^{f_y,g_y}(K,M)$:	c-iMD$^{f_y,g_c}(K,M)$:
$M_1 ... M_\ell \leftarrow Pad_s(M)$	$M_1 ... M_\ell \leftarrow Pad_s(M)$	$M_1 ... M_\ell \leftarrow Pad_s(M)$
$y_0 = IV$	$y_0 = IV$	$y_0 = IV$
for $i = 1$ **to** ℓ **do**	**for** $i = 1$ **to** ℓ **do**	**for** $i = 1$ **to** ℓ **do**
$y_i = f_x(g_x(K, M_i), y_{i-1})$	$y_i = f_y(M_i, g_y(K, y_{i-1}))$	$y_i = g_c(K, f_y(M_i, y_{i-1}))$
return $y = g_x(K, 0^{m-n} \| \| y_\ell)$	**return** $y = g_y(K, y_\ell)$	**return** $y = g_c(K, y_\ell)$

Figure 1: Pseudocode for the x-iMD, y-iMD, c-iMD constructions

Formally, x-iMD^{f_x,g_x}, y-iMD^{f_y,g_y}, c-iMD^{f_c,g_c} : $\{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$, where f_x, f_y, f_c : $\{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$, $g_x : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m$, $g_y, g_c : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$. The integration function can be any sufficiently mixing function, but for the sake of completeness, we suggest adopting the prepend-permute-chop (PPC) paradigm, which was proven indifferentiable from $\mathcal{RO}$ in [10]. Formally, $g(K, c) = [\pi(K||c)]^n$, where π is a random permutation, and the notation $[X]^n$ extracts the n most significant bits of the string X.

3 The Indifferentiability Framework

Proofs in the standard model (where adversarial resources are limited) can become extremely difficult for sufficiently complex cryptosystem. This fact motivated Bellare and Rogaway [4] to propose a formalism of the well known Random Oracle Model (ROM) which assumes the presence of a publicly accessible ideal primitive that when given an input, returns a random output. However, recent separation results [6, 14, 2, 7, 12, 11] questioned the soundness of this model since, in practice, $\mathcal{RO}$s are being instantiated by hash functions, which may not always behave like $\mathcal{RO}$s. Consequently, based on the notion of indifferentiability by Maurer *et al.* [12], in [8] Coron *et al.* introduced their hash function indifferentiability framework where a hash function is proven indifferentiable from $\mathcal{RO}$ and thus is expected to behave like one. In this framework, a distinguisher D (which plays the role of an adversary) is given oracle access to two (separate) systems, we call the first system the *real system*, consisting of a compression function and the construction under consideration (which needs to be proven indifferentiable from $\mathcal{RO}$), and we call the second system the *ideal system*, consisting of a $\mathcal{RO}$ and a simulator

[1] Instead of giving x-iMD a dedicated function to handle the last finalizing call, we use its own integration function g_x and pads its input with m-n 0 bits. This shouldn't affect its security arguments, but will simplify the indifferentiability proof. We don't have this issue with y-iMD and c-iMD.

(where the latter simulates the behaviour of the compression function of the real system at the ideal system). Indifferentiability proofs can be conducted in either the Random Oracle Model (ROM) or the Ideal Cipher Model (ICM). The difference between these two approaches is that in the ROM, the ideal compression function at the real system is modelled as a $\mathcal{RO}$, while it is modelled as an ideal block-cipher in the ICM. In the latter case, the ideal block-cipher can receive both forward and inverse queries (because block-ciphers are invertible). Regardless of the adopted model, proofs in the indifferentiability framework proceed in two steps: first, we propose the simulator (simulating a compression function or an ideal cipher), and then we prove that D's view is similar when it interacts with the real system as it is when it interacts with the ideal system (D cannot distinguish between the two systems). The formal definition of the indifferentiability framework is as follows [8] (where the constructions under consideration is referred to as a Turing machine C, the ideal compression function as $\mathcal{H}$ and the random oracle as $\mathcal{F}$):

Definition 3.1 (Indifferentiability from $\mathcal{RO}$)**.** *A Turing machine C with oracle access to an ideal primitive $\mathcal{H}$ is said to be (t_D, t_S, q, ϵ)-indifferentiable from an ideal primitive $\mathcal{F}$ if there exists a simulator S such that for any distinguisher D it holds that:*

$$|\Pr[D^{C,\mathcal{H}} = 1] - \Pr[D^{\mathcal{F},S} = 1]| < \epsilon$$

The simulator S has oracle access to $\mathcal{F}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and makes at most q queries to $C, \mathcal{H}, \mathcal{F}$ or S. C is said to be (computationally) indifferentiable from $\mathcal{F}$ if ϵ is a negligible function of the security parameter.

4 Indifferentiability of the iMD Constructions

In this section, we prove that the x-iMD^{f_x,g_x}, y-iMD^{f_y,g_y}, c-iMD^{f_c,g_c} constructions are indifferentiable from $\mathcal{RO}$ in the Ideal Cipher Model (ICM), when the compression functions f_x, f_y, f_c and the integration functions g_x, g_y, g_c are modelled as ideal block-ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c, \mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$, respectively. We adopt the ICM because most of the popular hash functions are (explicitly or implicitly) based on block-ciphers. The constructions x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, x-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, x-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$ treat $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ as black-boxes such that upon receiving a message M, they partition it into equally sized blocks $x_1, \ldots, x_\ell$ and process each block in turn through $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$. Formally, in c-iMD, $\mathcal{H}_c$ receives a message block $x_i \in \{0,1\}^m$ and a chaining variable (or IV) $y_i \in \{0,1\}^n$, and returns a temporary variable $c_i \in \{0,1\}^n$, which is immediately given to $\mathcal{G}_c$ along with the key $K \in \{0,1\}^k$ to finally return $z_i \in \{0,1\}^n$ (note that for a sequence of consecutive blocks, $y_{i+1} = z_i$). In x-iMD, on the other hand, the integration function $\mathcal{G}_x$ is called before $\mathcal{H}_x$ to process K and x_i which will return c_i that, along with y_i, will be given to $\mathcal{H}_x$ to return z_i. y-iMD is similar to x-iMD, except in this case $\mathcal{G}_y$ processes K with y_i instead of x_i, where the latter is given to $\mathcal{H}_y$ along with c_i (the output of $\mathcal{G}_y$) to produce z_i.

In the ideal system, we introduce two simulators, the compression function and the integration function simulators $S^{\mathcal{F}}, R^{\mathcal{F}}$, both with oracle access to the random oracle $\mathcal{F}$. Figure 2 depicts the indifferentiability games for the x-iMD, y-iMD, c-iMD constructions, showing how the distinguisher D accesses each system (where x-iMD, y-iMD, c-iMD are by $\hat{C}_x, \hat{C}_y, \hat{C}_c$, respectively).

4.1 The Distinguisher

The distinguisher D is an adversary with oracle access to two systems (the real and ideal systems) and whose aim is to prove that these systems can be distinguished from each other (but not

necessarily finding out which system is which, rather all D aims for is to show that the two systems behave differently). Precisely, D is given a *blind* oracle access to the systems, meaning that there is an imaginary *barrier* between D and the systems preventing D from seeing the systems, as depicted in figure 2. Trying to fool the systems, D carries out a set of tests on one system by repeatedly (and strategically) querying that system's different components and observes the responses, D sets the success conditions and outputs 1 if these tests succeed, 0 otherwise[2]. Simultaneously, D carries out the same set of tests on the other system, observes the responses and similarly outputs 1 or 0. D fails if both systems behaved consistently and then the two systems are said to be indifferentiable from each other. D may send either forward or inverse queries but format them differently depending on which component they are sent to; thus, D indeed knows what type of components it is interacting with (e.g. compression function, integration function etc.) but it does not know to which system they belong. In some cases, D can choose to exclusively interact with a particular system for a period of time, but it will not be able to choose which system that would be. Forward queries may be sent to any component in any system, but inverse queries may only be sent to $\mathcal{H}_i, \mathcal{G}_i, S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$; these components can receive inverse queries because we model the compression and the integration functions as ideal block-ciphers where these are clearly invertible. When D sends an inverse query to a particular component, it interacts with an inverse variant of that component A^{-1}, where $A \in \{\mathcal{H}_i, \mathcal{G}_i, S_i^{\mathcal{F}}, R_i^{\mathcal{F}}\}$. D communicates with the systems through three query channels, Ch_1, Ch_2, Ch_3, which are connected to three interfaces, if_1, if_2, if_3, at the barrier separating D from the real and ideal systems.

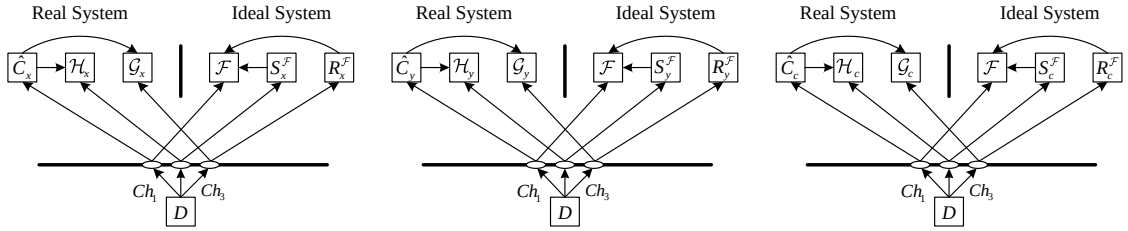
- Ch_1: D uses this channel to interact with the constructions $\hat{C}_i^{\mathcal{H}_i,\mathcal{G}_i}, i \in \{x, y, c\}$ and the random oracle $\mathcal{F}$. This channel supports only forward queries of the form (K, M), where $K \in \{0,1\}^k$ and $M \in \{\{0,1\}^m\}^*$, and delivers back to D the response $z \in \{0,1\}^n$. To simplify the proof, we assume that $|M|$ is a multiple of m, we also ignore padding rules, but the proof still holds with padding included.

- Ch_2: D uses this channel to interact with the ideal compression functions $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and their simulators $S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$. It supports both forward and inverse queries of the form $(\rightarrow, c, y)$ and $(\leftarrow, z, c)$ for $S_x^{\mathcal{F}}$, $(\rightarrow, x, c)$ and $(\leftarrow, z, x)$ for $S_y^{\mathcal{F}}$, and $(\rightarrow, x, y)$ and $(\leftarrow, c, x)$ for $S_c^{\mathcal{F}}$, then delivers back to D the appropriate responses, namely, it returns z for queries $(\rightarrow, c, y)$, $(\rightarrow, x, c)$, y for queries $(\leftarrow, z, c)$, $(\leftarrow, c, x)$, and c for query $(\leftarrow, z, x)$,$(\rightarrow, x, y)$.

- Ch_3: D uses this channel to communicate with the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ and their simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$. Like Ch_2, this channel also allows both forward and inverse queries of the form $(\rightarrow, x, K)$ and $(\leftarrow, c, K)$ for $R_x^{\mathcal{F}}$, $(\rightarrow, y, K)$ and $(\leftarrow, c, K)$ for $R_y^{\mathcal{F}}$, and $(\rightarrow, c, K)$ and $(\leftarrow, z, K)$ for $R_c^{\mathcal{F}}$, then delivers back to D the appropriate responses: x for query $(\leftarrow, c, K)$, y for query $(\leftarrow, c, K)$, c for queries $(\rightarrow, x, K)$,$(\rightarrow, y, K)$,$(\leftarrow, z, K)$, and z for query $(\rightarrow, c, K)$.

Each channel is split into two channels past its corresponding interface, one split for the real system and another for the ideal system. Hence, when an interface if_x receives a query from D, it creates two identical copies of that query and sends them off the other end through the channel's two splits, unless D chooses to exclusively interacts with a single system for a period of time, in which case the interface chooses that system at random (D cannot choose which system to interact with) and starts sending D's queries to the components of that system until D advises otherwise. That is, the interfaces cannot randomly switch between the systems without an explicit request from D.

[2]In other words, D outputs 1 if it thinks that it is interacting with, e.g., the idea system, otherwise it outputs 0 (or vice versa depending on D's definition).



(a) D's view in the x-iMD game (b) D's view in the y-iMD game (c) D's view in the c-iMD game

Figure 2: The interaction between D and the real/ideal systems in the x-iMD, y-iMD, c-iMD games

4.2 The Indifferentiability Proof

In this section, we adopt the Ideal Cipher Model (ICM) and prove that the constructions x-iMD, y-iMD, c-iMD, with access to the ideal ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c, \mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$ are indifferentiable from a $\mathcal{RO}$, except with negligible probability.

Theorem 4.1 (Indifferentiability of x-iMD, y-iMD, c-iMD)**.** *The block-cipher based Integrated-key constructions x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, y-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, c-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$, with oracle access to the ideal block-ciphers $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c : \{0,1\}^m \times \{0,1\}^n \rightarrow \{0,1\}^n$, and the ideal integration functions $\mathcal{G}_x : \{0,1\}^k \times \{0,1\}^m \rightarrow \{0,1\}^m, \mathcal{G}_y, \mathcal{G}_c : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$, modelled as ideal block-ciphers, are $(t_D, t_S, q_1, q_2, q_3, \epsilon_x)$-indifferentiable $(t_D, t_S, q_1, q_2, q_3, \epsilon_y)$-indifferentiable $(t_D, t_S, q_1, q_2, q_3, \epsilon_c)$-indifferentiable from a random oracle $\mathcal{F}$ in the ideal cipher model for $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$, any t_D, $t_S \leq (q_1 \cdot L/m + q_2 + q_3) \cdot (m+n)$, any number of queries q_1, q_2, q_3 sent by D to x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, y-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, c-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$, and:*

$$
\begin{aligned}
\epsilon_x &\leq \left(2^n((q_1.L/m + q_2 + q_3) + 2(q_1.L/m)^2 + q_2 + q_3) + (q_1.L/m + q_2 + q_3)\right)/2^n \\
&+ \left(4(q_1.L/m + q_2 + q_3) + 3(q_1.L/m + q_2 + q_3)^2\right)/2^n \\
\epsilon_y &\leq \left(2^m(2(q_1.L/m + q_2 + q_3) + 2(q_1.L/m + q_2 + q_3)^2) + (q_1.L/m + q_2 + q_3)\right)/2^{m+n} \\
&+ \left(3(q_1.L/m + q_2 + q_3) + 32(q_1.L/m + q_2 + q_3)^2\right)/2^n + (q_1.L/m + q_2 + q_3)/2^m \\
\epsilon_c &\leq \left(2^m(3(q_1.L/m + q_2 + q_3) + 2(q_1.L/m + q_2 + q_3)^2) + (q_1.L/m + q_2 + q_3)\right)/2^{m+n} \\
&+ \left(2(q_1.L/m + q_2 + q_3) + 3(q_1.L/m + q_2 + q_3)^2\right)/2^n
\end{aligned}
$$

where L is the maximum length of the query q_1.

Proof. We prove the indifferentiability by means of a hybrid argument. We adopt the game-playing approach [5, 8] and prove that consecutive games are indifferentiable from each other, stating the distinguishing probability when applicable. Each game represents a state of the system which then evolves as the proof progresses through the games. We start with $G(1)$, Game 1, which represents the ideal system (consisting of the $\mathcal{RO}$ and simulators of the ideal compression and integration functions) and finish with $G(8)$ (consisting of the construction and the ideal compression and integration functions), Game 8, which represents the real system. We denote x-iMD$^{\mathcal{H}_x,\mathcal{G}_x}$, y-iMD$^{\mathcal{H}_y,\mathcal{G}_y}$, c-iMD$^{\mathcal{H}_c,\mathcal{G}_c}$ by $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$, respectively, and prove that they are indifferentiable from the random oracle $\mathcal{F}$. Below we integrate the indifferentiability proofs of the three constructions noting that the real/ideal systems of each proof may consist of slightly different components, which we will often state explicitly.

The Simulators. We first propose the required simulators. The proof requires a total of six simulators proposed in the ideal systems to simulate components in the real systems. Simulators

$S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$ simulate the ideal compression functions $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$ and simulators $R_x^{\mathcal{F}}, R_y^{\mathcal{F}}, R_c^{\mathcal{F}}$ simulate the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$, respectively; all simulators have oracle access to $\mathcal{F}$. The proof for $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}$ uses $S_x^{\mathcal{F}}, R_x^{\mathcal{F}}$, the proof for $\hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}$ uses $S_y^{\mathcal{F}}, R_y^{\mathcal{F}}$, and the proof for $\hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ uses $S_c^{\mathcal{F}}, R_c^{\mathcal{F}}$. Figure 4.2 graphically illustrates the input/output notation for each construction, we will use this notation extensively throughout the proof. Each simulator pair

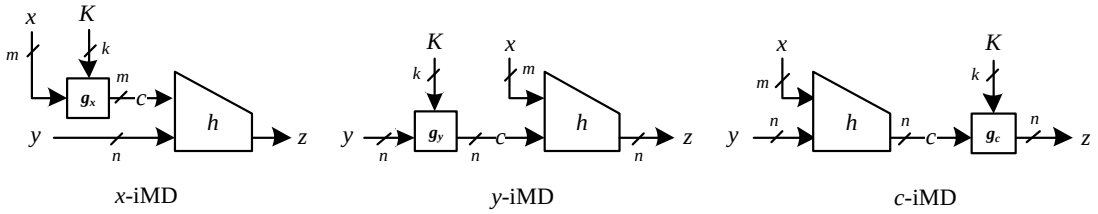


Figure 3: Input/output notation of the x-iMD, y-iMD, c-iMD constructions

$S_i^{\mathcal{F}}$-$R_i^{\mathcal{F}}$ cooperatively maintains a table $\mathcal{T}_i^K$, that is initially empty $\perp$ but gradually grow as D interacts with $S_i^{\mathcal{F}}$-$R_i^{\mathcal{F}}$, where $i \in \{x, y, c\}$. Since integrated-key hash functions are actually families of hash functions, where the individual function members are indexed by different keys $K \in \mathcal{K}$, $S_i^{\mathcal{F}}$ and $R_i^{\mathcal{F}}$ will maintain different tables $\mathcal{T}_i^K$ for different keys (members). Without loss of generality, here we will assume that we are interacting with a single hash function member and that the key K is fixed throughout the proof. As illustrated in figure 4, all tables contain 5-tuple records of the form $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$, where $x_i \in \{0,1\}^m, y_i \in \{0,1\}^n$ are the message block and chaining variable (or IV), $z_i \in \{0,1\}^n$ is the output of the ideal ciphers $\mathcal{H}_x, \mathcal{H}_y$ (in the case of x-iMD and y-iMD) or the output of the idea integration function $\mathcal{G}_c$ (in the case of c-iMD), and c_i is the output of the ideal integration functions $\mathcal{G}_x, \mathcal{G}_y$ (in the case of x-iMD and y-iMD) or the output of the ideal cipher $\mathcal{H}_c$ (in the case of c-iMD), such that:

$$c_i \in \begin{cases} \{0,1\}^m & \text{if } c_i \in \mathcal{T}_x^K \\ \{0,1\}^n & \text{if } c_i \in \mathcal{T}_y^K \cup \mathcal{T}_c^K \end{cases}$$

The index $i \in \{0, 1, \dots\}$ of a record specifies the location of the tuple in tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, the tag $t_i \in \{\perp, 0, 1\}$ indicates whether the tuple is part of a sequence, and if it does s_l indicates to which sequence (indexed by $l \in \{\perp, 0, 1, \dots\}$) this tuple belongs, and $p \in \{\perp, 0, 1, \dots\}$ specifies the exact location of the tuple in the sequence s_l. A sequence is an ordered list of tuples such that $z_a^{s_l,p-1} = y_b^{s_l,p}$, and is rooted by the j-th tuple where $y_j^{s_l,0} = IV$; note that if two consecutive tuples in a sequence do not have to be consecutive in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ (if $z_a^{s_l,p-1} = y_b^{s_l,p}$, then "$a$" does not have to be "$b-1$", while they are still indexed in succession in the sequence s_l). A tuple belonging to a sequence is called *sequenced* tuple, otherwise it is *singular*. To keep track of the number of sequences, $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ maintain counters $\tilde{C}_x^K, \tilde{C}_y^K, \tilde{C}_c^K$ which are initially set to 0 but are incremented every time a tuple with $y_\_ = IV$ is encountered (recall that we use the notation $\_$ (underscore) to denote an arbitrary tuple in a table). As per the definitions of $S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$ below, a sequence has to be rooted by a tuple where $y_\_^{s_l,0} = IV$, otherwise a sequence may not be formed, even if there are tuples in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ that can be connected. Clearly, $t = \perp$ implies $l = p = \perp$; in fact, t here acts as a switch to activate or deactivate the index s_l (however, we may sometimes drop the s_l, p index when referencing tuples if it is not needed). Furthermore, tuples in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ are required to be complete (there are no tuples with missing fields). Figures 5 and 6 formally define simulators $S_i^{\mathcal{F}}, R_i^{\mathcal{F}}, i \in \{x, y, c\}$ and their inverse variants; these simulators are based on two simple rule:

1. The c value is always generated by a $\mathcal{RO}$. If it was given in a query, c is further processed through a $\mathcal{RO}$, that is $\mathcal{F}(c)$.

2. Unless given in a query, the z value is always generated by a $\mathcal{RO}$.

Below we describe the simulators in more details, recall that all simulators receive both forward and inverse queries.

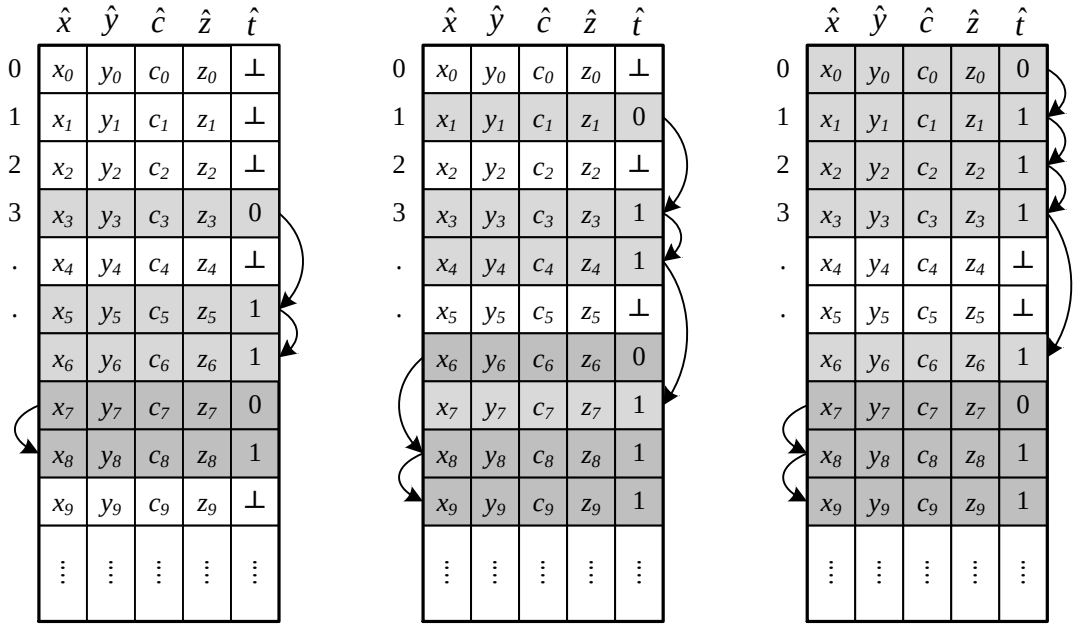


Figure 4: Samples of tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ illustrating how tuples and sequences are organised and linked

Simulators $S_x^{\mathcal{F}}$ and $(S_x^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, c, y)$, $S_x^{\mathcal{F}}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$ and $y_i = y$, if found, it returns z_i. Otherwise, $S_x^{\mathcal{F}}$ generates a new value for x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, then $S_x^{\mathcal{F}}$ proceeds to generate z as follows:

   1(a). If $y = IV$, then $S_x^{\mathcal{F}}$ makes the query $\mathcal{F}(\mathcal{F}(c), y)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1, p = 0$.

   1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_x^K$ such that $y = z_i^{s_l,p}$, then $S_x^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(\mathcal{F}(c), y)$, while setting $t = 1$ and indexes the new tuple by $s_l, p + 1$.

   1(c). Otherwise, $S_x^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(\mathcal{F}(c), y)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, z, c)$, $(S_x^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$ and $c_i = c$, if found, it returns y_i. Otherwise, $(S_x^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, where $\mathcal{T}_x^K(\hat{y})$ and $\mathcal{T}_x^K(\hat{z})$ extract all the y and z values in the table $\mathcal{T}_x^K$ which, along with IV, will be excluded from the value assigned to y, and similarly generates x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, while setting $t = \perp$, with s_l deactivated.

Simulators $R_x^{\mathcal{F}}$ and $(R_x^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, x)$, $R_x^{\mathcal{F}}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$, if found, it returns c_i. Otherwise, $R_x^{\mathcal{F}}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, then query $\mathcal{F}$ on x to obtain c, that is $c \leftarrow \mathcal{F}(K, x)$, and finally query $\mathcal{F}$ on y and c to obtain z, that is $z \leftarrow \mathcal{F}(y, c)$, while setting $t = \perp$ with s_l deactivated.

2. On inverse query $(\leftarrow, K, c)$, $(R_x^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_x^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns x_i. Otherwise, $(R_x^{\mathcal{F}})^{-1}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$, and then generates z by querying $\mathcal{F}$ that is $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$. Finally, it sets $t = \perp$ with the index s_l deactivated.

Simulators $S_y^{\mathcal{F}}$ and $(S_y^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, x, c)$, $S_y^{\mathcal{F}}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $c_i = c$, if found, it returns z_i. Otherwise, $S_y^{\mathcal{F}}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$. Then $S_y^{\mathcal{F}}$ obtains z by querying $\mathcal{F}$ as follows: $z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$, while setting $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, z, x)$, $(S_y^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$ and $x_i = x$, if found, it returns c_i. Otherwise, $(S_y^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$, and queries $\mathcal{F}$ to obtain c such that $c \leftarrow \mathcal{F}(K, y)$, while setting $t = \perp$, with s_l deactivated.

Simulators $R_y^{\mathcal{F}}$ and $(R_y^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, y)$, $R_y^{\mathcal{F}}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $y_i = y$, if found, it returns c_i. Otherwise, $R_y^{\mathcal{F}}$ generates x uniformly at random $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$, and query $\mathcal{F}$ to generate $c \leftarrow \mathcal{F}(K, y)$, then $R_y^{\mathcal{F}}$ proceeds to generate z as follows (recall that z has to be generated to complete the new tuple):

   1(a). If $y = IV$, then $R_y^{\mathcal{F}}$ makes the query $\mathcal{F}(x, c)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1$ (increment the counter $\tilde{C}$) and $p = 0$.

   1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_c^K$ such that $y = z_i^{s_l,p}$, then $R_y^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(T||x, c)$, where:

$$T = x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p}$$

   which is a chain of queries rooted by the tuple indexed by $s_l, 0$. Once the new tuple is created, $R_y^{\mathcal{F}}$ indexes it by $s_l, p+1$ while setting $t^{s_l,p+1} = 1$.

   1(c). Otherwise, $R_y^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(x, c)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, K, c)$, $(R_y^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_y^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns y_i. Otherwise, $(R_y^{\mathcal{F}})^{-1}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$, and then generates z by querying $\mathcal{F}$ such that $z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$. Finally, it sets $t = \perp$ with the index s_l deactivated.

Simulators $S_c^{\mathcal{F}}$ and $(S_c^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, x, y)$, $S_c^{\mathcal{F}}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $y_i = y$, if found, it returns c_i. Otherwise, $S_c^{\mathcal{F}}$ creates a new tuple by querying $\mathcal{F}$ for c on input (x, y), that is $c \leftarrow \mathcal{F}(x, y)$, then $S_c^{\mathcal{F}}$ proceeds to generate z as follows (recall that z has to be generated to complete the new tuple):

   1(a). If $y = IV$, then $S_c^{\mathcal{F}}$ makes the query $\mathcal{F}(K, c)$ to the random oracle $\mathcal{F}$ to obtain z, and sets $t = 0$, $l = \tilde{C}_K + 1$ (increment the counter $\tilde{C}$) and $p = 0$.

   1(b). If there is a tuple $(x_i^{s_l,p}, y_i^{s_l,p}, c_i^{s_l,p}, z_i^{s_l,p}, t_i^{s_l,p})$ in $\mathcal{T}_c^K$ such that $y = z_i^{s_l,p}$, then $S_c^{\mathcal{F}}$ obtains z by querying the random oracle $\mathcal{F}(K, T||x)$, where:

$$T = x_{\_}^{s_l,0}||x_{\_}^{s_l,1}||\ldots||x_{\_}^{s_l,p-1}||x_{\_}^{s_l,p}$$

   which is a chain of queries rooted by the tuple indexed by $s_l, 0$. Once the new tuple is created, $S_c^{\mathcal{F}}$ indexes it by $s_l, p+1$ while setting $t^{s_l,p+1} = 1$.

   1(c). Otherwise, $S_c^{\mathcal{F}}$ obtains z by querying $\mathcal{F}(K, c)$, and sets $t = \perp$, with s_l deactivated.

2. On inverse query $(\leftarrow, c, x)$, $(S_c^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $x_i = x$ and $c_i = c$, if found, it returns y_i. Otherwise, $(S_c^{\mathcal{F}})^{-1}$ generates y uniformly at random $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, where $\mathcal{T}_c^K(\hat{y})$ and $\mathcal{T}_c^K(\hat{z})$ extract all the y and z values in the table $\mathcal{T}_c^K$ which, along with IV, will be excluded from the value assigned to y, obtains z by querying $\mathcal{F}(K, \mathcal{F}(c))$, and sets $t = \perp$, with s_l deactivated.

Simulators $R_c^{\mathcal{F}}$ and $(R_c^{\mathcal{F}})^{-1}$

1. On forward query $(\rightarrow, K, c)$, $R_c^{\mathcal{F}}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $c_i = c$, if found, it returns z_i. Otherwise, $R_c^{\mathcal{F}}$ generates both x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, and then queries $\mathcal{F}$ to generate $z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$. Finally, it sets $t = \perp$ with the index s_l deactivated.

2. On inverse query $(\leftarrow, K, z)$, $(R_c^{\mathcal{F}})^{-1}$ searches $\mathcal{T}_c^K$ for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $z_i = z$, if found, it returns c_i. Otherwise, $(R_c^{\mathcal{F}})^{-1}$ generates x and y uniformly at random: $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$, $y \xleftarrow{\$} \{0,1\}^n \backslash \{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$, and then generates c by querying $\mathcal{F}$ on input (x, y): $c \leftarrow \mathcal{F}(x, y)$. Finally, it sets $t = \perp$ with the index s_l deactivated.

The Indifferentiability Proof. We now construct the games. Throughout the proof, P_i^x, P_i^y, P_i^c denote the probabilities that D outputs 1 in game $G(i)$ of the indifferentiability proofs of $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$, respectively. Figure 7 depicts the state of the systems in each game.

Game 1. This is the $\mathcal{RO}$ game where D is exclusively interacting with the ideal system. Let

$$P_1^x = \Pr[D^{\mathcal{F},S_x^{\mathcal{F}},R_x^{\mathcal{F}}} = 1]; P_1^y = \Pr[D^{\mathcal{F},S_y^{\mathcal{F}},R_y^{\mathcal{F}}} = 1]; P_1^c = \Pr[D^{\mathcal{F},S_c^{\mathcal{F}},R_c^{\mathcal{F}}} = 1]$$

Game 2. In this game we introduce dummy relay algorithms $F_{1,x}, F_{1,y}, F_{1,c}$ placed between D and $\mathcal{F}$ in the $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ proofs, respectively. These algorithms relay queries from D to $\mathcal{F}$ and relay responses back from $\mathcal{F}$ to D. Obviously, the view of D in $G(2)$ is not affected by the introduction of $F_{1,x}, F_{1,y}, F_{1,c}$, thus:

$$P_2^x = \Pr[D^{F_{1,x}^{\mathcal{F}},S_x^{\mathcal{F}},R_x^{\mathcal{F}}} = 1] = P_1^x; P_2^y = \Pr[D^{F_{1,y}^{\mathcal{F}},S_y^{\mathcal{F}},R_y^{\mathcal{F}}} = 1] = P_1^y; P_2^c = \Pr[D^{F_{1,c}^{\mathcal{F}},S_c^{\mathcal{F}},R_c^{\mathcal{F}}} = 1] = P_1^c$$

Simulator $S_x^{\mathcal{F}}(\rightarrow, c, y)$
if $(c_i, y_i) \in \mathcal{T}_x^K$
$\wedge(c, y) = (c_i, y_i)$
then return z_i
else $x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$
if $y = \text{IV}$ **then**
$t = 0,\ l = \tilde{C}_k + 1,\ p = 0$
return $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$
else if $y = z_i^{s_l,p}, z_i^{s_l,p} \in \mathcal{T}_x^K$
then set $t = 1, p = p + 1$
return $z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$
else $z \leftarrow \mathcal{F}(\mathcal{F}(c), y),\ t = \perp$

Simulator $(S_x^{\mathcal{F}})^{-1}(\leftarrow, z, c)$
if $(z_i, c_i) \in \mathcal{T}_x^K$
$\wedge(z, c) = (z_i, c_i)$
return y_i
else return $y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\},\ t = \perp$

Simulator $S_y^{\mathcal{F}}(\rightarrow, x, c)$
if $(x_i, c_i) \in \mathcal{T}_y^K$
$\wedge(x, c) = (x_i, c_i)$
then return z_i
else $y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
return
$z \leftarrow \mathcal{F}(x, \mathcal{F}(c))$
$t = 0$

Simulator $(S_y^{\mathcal{F}})^{-1}(\leftarrow, z, x)$
if $(z_i, x_i) \in \mathcal{T}_y^K$
$\wedge(z, x) = (z_i, x_i)$
then return c_i
else
$y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
return $c \leftarrow \mathcal{F}(K, y)$
$t = \perp$

Simulator $S_c^{\mathcal{F}}(\rightarrow, x, y)$
if $(x_i, y_i) \in \mathcal{T}_c^K$
$\wedge(x, y) = (x_i, y_i)$
return c_i
else return $c \leftarrow \mathcal{F}(x, y)$
if $y = \text{IV},$ **then**
$t = 0,\ l = \tilde{C}_k + 1,$
$p = 0,\ z \leftarrow \mathcal{F}(K, c)$
else if $y = z_i^{s_l,p}$
$\wedge z_i^{s_l,p} \in \mathcal{T}_c^K,$ **then**
$t = 1, p = p + 1$
$T = x^{s_l,0}_{\_} || x^{s_l,1}_{\_} || \ldots || x^{s_l,p}_{\_}$
$z \leftarrow \mathcal{F}(K, T||x)$
else $z \leftarrow \mathcal{F}(K, c),\ t = \perp$

Simulator $(S_c^{\mathcal{F}})^{-1}(\leftarrow c, x)$
if $(c_i, x_i) \in \mathcal{T}_c^K$
$\wedge(c, x) = (c_i, x_i),$ **return** y_i
else return $y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
$z \leftarrow \mathcal{F}(K, \mathcal{F}(c)),\ t = \perp$

Figure 5: Simulators $S_x^{\mathcal{F}}, (S_x^{\mathcal{F}})^{-1}, S_y^{\mathcal{F}}, (S_y^{\mathcal{F}})^{-1}, S_c^{\mathcal{F}}, (S_c^{\mathcal{F}})^{-1}$

Simulator $R_x^{\mathcal{F}}(\rightarrow, K, x)$
if $x_i \in \mathcal{T}_x^K \wedge x = x_i$
return c_i
else $t = \perp$
$c \leftarrow \mathcal{F}(K, x)$
$y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
return
$z \leftarrow \mathcal{F}(y, c)$

Simulator $(R_x^{\mathcal{F}})^{-1}(\leftarrow, K, c)$
if $c_i \in \mathcal{T}_x^K \wedge c = c_i$
return x_i
else $t = \perp$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_x^K(\hat{x})\}$
$y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_x^K(\hat{z}) \cup IV\}$
return
$z \leftarrow \mathcal{F}(\mathcal{F}(c), y)$

Simulator $R_y^{\mathcal{F}}(\rightarrow, K, y)$
if $y_i \in \mathcal{T}_y^K \wedge y = y_i,$ **return** c_i
else return $c \leftarrow \mathcal{F}(K, y)$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$
if $y = IV$ **then**
$t = 0,\ l = \tilde{C}_k + 1,$
$p = 0,\ z \leftarrow \mathcal{F}(x, c)$
else if $y = z_i^{s_l,p}$
$\wedge z_i^{s_l,p} \in \mathcal{T}_y^K$ **then**
$t = 1, p = p + 1$
$T = x^{s_l,0}_{\_} || x^{s_l,1}_{\_} || \ldots || x^{s_l,p}_{\_}$
$z \leftarrow \mathcal{F}(T||x, c)$
else $z \leftarrow \mathcal{F}(x, c),\ t = \perp$

Simulator $(R_y^{\mathcal{F}})^{-1}(\leftarrow, K, c)$
if $c_i \in \mathcal{T}_y^K \wedge c = c_i,$ **return** y_i
else return $y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{z}) \cup IV\}$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_y^K(\hat{x})\}$
$z \leftarrow \mathcal{F}(x, \mathcal{F}(c)),\ t = \perp$

Simulator $R_c^{\mathcal{F}}(\rightarrow, K, c)$
if $c_i \in \mathcal{T}_c^K \wedge c = c_i$
return z_i
else $t = \perp$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\})$
$y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
return
$z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$

Simulator $(R_c^{\mathcal{F}})^{-1}(\leftarrow, K, z)$
if $z_i \in \mathcal{T}_c^K \wedge z = z_i$
return c_i
else $t = \perp$
$x \xleftarrow{\$} \{0,1\}^m \backslash \{\mathcal{T}_c^K(\hat{x})\}$
$y \xleftarrow{\$} \{0,1\}^n \backslash$
$\{\mathcal{T}_c^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{z}) \cup IV\}$
return
$c \leftarrow \mathcal{F}(x, y)$

Figure 6: Simulators $R_x^{\mathcal{F}}, (R_x^{\mathcal{F}})^{-1}, R_y^{\mathcal{F}}, (R_y^{\mathcal{F}})^{-1}, R_c^{\mathcal{F}}, (R_c^{\mathcal{F}})^{-1}$

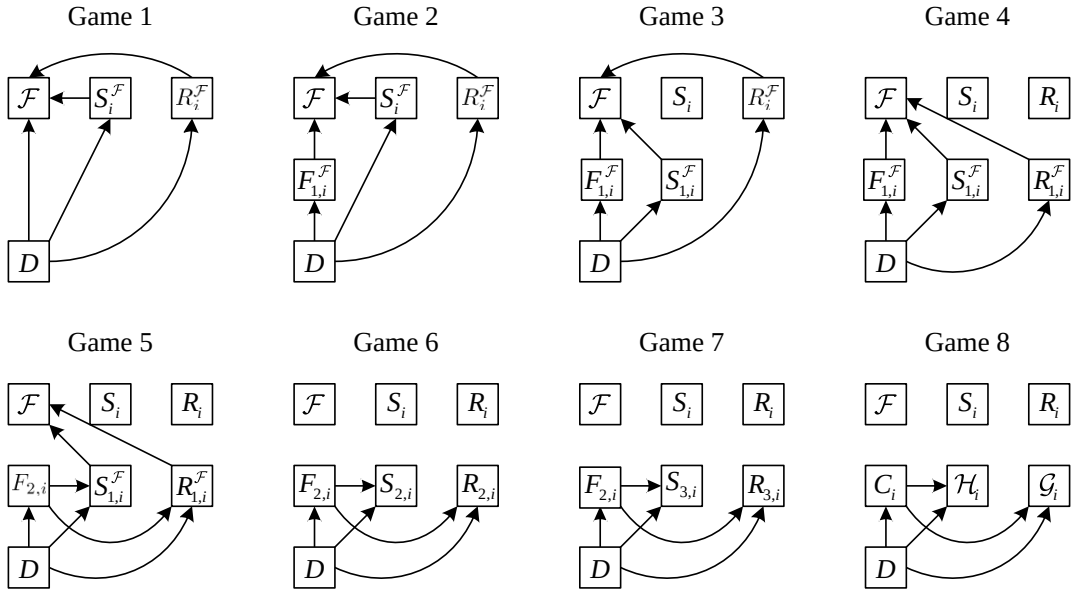


Figure 7: A depiction of how the system's state evolves through the games in the indifferentiability proof of $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$

Game 3. In this game we introduce a slightly modified replicas of $S_x^{\mathcal{F}}, S_y^{\mathcal{F}}, S_c^{\mathcal{F}}$ (and their inverse variants), still with oracle access to $\mathcal{F}$. D will now interact with the modified simulators $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$ which, upon receiving a query, create a new tuple (x, y, c, z, t) but explicitly abort if any of the following failure conditions is satisfied:

1. On forward queries $S_{1,x}^{\mathcal{F}}(\rightarrow, c, y), S_{1,y}^{\mathcal{F}}(\rightarrow, x, c), S_{1,c}^{\mathcal{F}}(\rightarrow, x, y)$, the simulators create the new tuple (x, y, c, z, t), but the following collisions occur:

   1(a). Fixed point: either in $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$, it is the case that $z = IV$, or in $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$, it is the case that $z = y$.

   1(b). Prefix collision: in $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}$, it is the case that $z = y_i$ for some $y_i \in \mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{y})$.

   1(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

      i. when $S_{1,x}^{\mathcal{F}}$ receives $(\rightarrow, c, y)$, the following hold: $z = z_i \wedge (c, y) \neq (c_i, y_i)$.

      ii. when $S_{1,y}^{\mathcal{F}}$ receives $(\rightarrow, x, c)$, the following hold: $z = z_i \wedge (x, c) \neq (x_i, c_i i)$.

      iii. when $S_{1,c}^{\mathcal{F}}$ receives $(\rightarrow, x, y)$, the following hold: $c = c_i \wedge (x, y) \neq (x_i, y_i)$, or $z = z_i \wedge c \neq c_i$.

2. On inverse queries $(S_{1,x}^{\mathcal{F}})^{-1}(\leftarrow, z, c), (S_{1,y}^{\mathcal{F}})^{-1}(\leftarrow, z, x), (S_{1,c}^{\mathcal{F}})^{-1}(\leftarrow, c, x)$, the simulators create the new tuple (x, y, c, z, t), but the following collisions occur:

   2(a). Fixed point: in $(S_{1,c}^{\mathcal{F}})^{-1}$, it is the case that $z = IV$.

   2(b). Prefix collision: in $(S_{1,x}^{\mathcal{F}})^{-1}, (S_{1,y}^{\mathcal{F}})^{-1}, (S_{1,c}^{\mathcal{F}})^{-1}$, it is the case that $z = y_i$ for some $y_i \in \mathcal{T}_x^K(\hat{y}) \cup \mathcal{T}_y^K(\hat{y}) \cup \mathcal{T}_c^K(\hat{y})$.

   2(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

      i. when $(S_{1,y}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, x)$, the following hold: $c = c_i \wedge y \neq y_i$.

      ii. when $(S_{1,c}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, c, x)$, the following hold: $z = z_i \wedge c \neq c_i$.

2(d). Partial query collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that:

i. when $(S_{1,x}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, c)$, the following hold: $c \neq c_i \wedge z = z_i$.

ii. when $(S_{1,y}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, z, x)$, the following hold: $x \neq x_i \wedge z = z_i$.

iii. when $(S_{1,c}^{\mathcal{F}})^{-1}$ receives $(\leftarrow, c, x)$, the following hold: $x \neq x_i \wedge c = c_i$.

Clearly, failure conditions 1(a) and 2(a) are similar. In this case, there are two types of fixed points, (i) when $z = IV$, and (ii) when $z = y$. Generating $z_i = IV$ may indeed happen but with very low probability since in the case of $S_{1,x}^{\mathcal{F}}$ and $S_{1,y}^{\mathcal{F}}$, there is only one combination of c and y or x and c, respectively, that will cause $z = IV$; similarly, in the case of $S_{1,c}^{\mathcal{F}}$, there is one value of c (combined with the fixed key K) that will cause $z = IV$. That is, unless given as part of a query, z is always generated by a $\mathcal{RO}$ (according to the definitions of the simulators), but while generating it, we cannot instruct the $\mathcal{RO}$ to exclude IV from the possible values it may return for z. Similarly, the fixed point $z = y$ (the output collides with the input) can only happen when querying $S_{1,x}^{\mathcal{F}}$ and $S_{1,c}^{\mathcal{F}}$. In both cases, y is given as part of the query, but we do not know what $\mathcal{RO}$ query would generate the given y, and since z (in both cases) is generated by a $\mathcal{RO}$, it may be the case that the inputs used to generate z are the ones that would generate y. Even though in $S_{1,x}^{\mathcal{F}}$ the y value is actually part of the $\mathcal{RO}$ input that generates z, there is nothing stopping the $\mathcal{RO}$ from output y (part of its input) as this is equally random. Clearly, none of the inverse simulators $(S_{1,x}^{\mathcal{F}})^{-1}, (S_{1,y}^{\mathcal{F}})^{-1}, (S_{1,c}^{\mathcal{F}})^{-1}$ may output $z = y$ since z is given as part of their queries (and in the case of $(S_{1,c}^{\mathcal{F}})^{-1}$, y is generated excluding the value of z). When calculating the probability of all these failure conditions we should account for queries sent to both $S_i^{\mathcal{F}}$ and $R_i^{\mathcal{F}}$ since they cooperatively add tuples to $\mathcal{T}_i^K$ whenever they are queried, where $i \in \{x, y, c\}$. However, since it is possible that new queries will not create new tuples (if they match existing tuples), the probability is upper bounded by $q_2 + q_3/2^n$, where q_2 are queries sent to $S_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}$ and q_3 are queries sent to $R_{1,i}^{\mathcal{F}}, (R_{i,i}^{\mathcal{F}})^{-1}$, where $i \in \{x, y, c\}$. We do not account for q_1 in this probability because the q_1 queries are sent to F_1, F_2, F_3 which, at this stage, do not contribute in updating the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, respectively, and so they will not affect the probability of finding collisions there.

Failure conditions 1(b) and 2(b) are similar. However, in this case z may collide with any y_i belonging to any tuple in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, but since not all queries will add new tuples to the table, this probability is upper bounded by the birthday attack [15], that is $\leq (q_2 + q_3)^2/2^n$. When creating a new tuple (x, y, c, z, t) upon receiving a query, there are two types of prefix collisions, either $y = z_i$ or $z = y_j$ for some $z_i, y_j \in \mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. In the first type, if z_i is sequenced (part of a sequence) and $y = z_i$, then the simulators connect the tuple of the newly queried y with the tuple of the matching z_i (the i-th tuple) and no collision occurs (recall that only simulators $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}$ can create/extend sequences because they are the only simulators who receive y as part of their queries). But what if z_i is not sequenced? In this case, the simulators will treat it as a sequenced tuple that is part of a sequence consisting of only one tuple, itself (the i-th tuple). Connecting the newly created tuple with the i-th tuple will create an "unrooted" sequence, one that has no route tuple with $y = IV$, but this will not affect the indifferentiability proofs, as we will see later. This leaves the second type of prefix collision, $z = y_j$, which applies to $S_{1,x}^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}$ and their inverse variants. To see why this is the case, we discuss how such collision may be generated in $\mathcal{T}_x^K$, which is maintained by $S_{1,x}^{\mathcal{F}}, (S_{1,x}^{\mathcal{F}})^{-1}, R_{1,x}^{\mathcal{F}}, (R_{1,x}^{\mathcal{F}})^{-1}$, similar argument apply for $\mathcal{T}_y^K$ and $\mathcal{T}_c^K$. Simulator $S_{1,x}^{\mathcal{F}}$ receives y as part of its query and then creates a new tuple if necessary, but other simulators have no way to exclude the y value that $S_{1,x}^{\mathcal{F}}$ received when they generate their z values as the latter is generated by a $\mathcal{RO}$ which may output the value of the previously queried y with probably $(q_2 + q_3)^2/2^n$.

Failure conditions 1(c) and 2(c), on the other hand, allude to a more fundamental problem when the simulators receive the c and z values from D rather then generating them. These values

should ideally be generated by a $\mathcal{RO}$, but on queries to $S^{\mathcal{F}}_{1,x}, (S^{\mathcal{F}}_{1,x})^{-1}, S^{\mathcal{F}}_{1,y}, (S^{\mathcal{F}}_{1,c})^{-1}, (R^{\mathcal{F}}_{1,x})^{-1}$, $(R^{\mathcal{F}}_{1,y})^{-1}, R^{\mathcal{F}}_{1,c}$, the value of c is chosen by D and is given as part of the query, similarly for z when D queries $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$. In these situations, since we cannot invert c and z to obtain the $\mathcal{RO}$ inputs that generated them, we generate these inputs uniformly at random (recall that when creating a new tuple, all fields of that tuple have to be generated, even if they were not part of the particular query that triggered the tuple to be created), but that does not mean that if these inputs were supplied to a $\mathcal{RO}$, it will return the corresponding c and z that were sent to the simulators by D earlier, this behaviour may lead to internal collisions. Below we analyse all internal collision scenarios corresponding to simulators $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ and their inverse variants (as outlined in failure conditions 1(c) and 2(c) above, respectively). In all scenarios, the simulators receive queries of a particular format (depending on the receiving simulator) from D and creates the new tuple (x, y, c, z, t).

1. $S^{\mathcal{F}}_{1,x}$: simulator $S^{\mathcal{F}}_{1,x}$ receives c as part of its query $(\rightarrow, c, y)$, so if c exists in $\mathcal{T}^K_x$, no new tuple will be created and thus no collision. However, when D sends the query $(\leftarrow, z_i, c_i)$ to $(S^{\mathcal{F}}_{1,x})^{-1}$, the latter has no way to know what value of y_i (in combination of the given c_i) would generate the given z_i, so it generates y_i uniformly at random, but it is likely that $\mathcal{F}(\mathcal{F}(c_i), y_i) \neq z_i$. If D later sends the query $(\rightarrow, c, y)$ to $S^{\mathcal{F}}_{1,x}$, the latter generates z by $z = \mathcal{F}(\mathcal{F}(c), y)$, a collision then occurs if $z = z_i$ while $(c, y) \neq (c_i, y_i)$.

2. $S^{\mathcal{F}}_{1,y}$: simulator $S^{\mathcal{F}}_{1,y}$ receives c as part of its query $(\rightarrow, x, c)$, so if c exists in $\mathcal{T}^K_y$, no new tuple will be created and thus no collision. However, when D sends the query $(\leftarrow, z_i, x_i)$ to $(S^{\mathcal{F}}_{1,y})^{-1}$, the latter has no way to know what value of c_i (in combination of the given x_i) would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(K, y_i)$, where y_i is generated uniformly at random, but it is likely that $\mathcal{F}(\mathcal{F}(c_i), y_i) \neq z_i$. If D later sends the query $(\rightarrow, x, c)$ to $S^{\mathcal{F}}_{1,y}$, the latter generates z by $z = \mathcal{F}(x, \mathcal{F}(c))$, a collision then occurs if $z = z_i$ while $(x, c) \neq (x_i, c_i)$.

3. $S^{\mathcal{F}}_{1,c}$: When D sends the queries $(\leftarrow, c_i, x_i)$ and $(\rightarrow, K, c_j)$ to $(S^{\mathcal{F}}_{1,c})^{-1}$ and $R^{\mathcal{F}}_{1,c}$, respectively, both $(S^{\mathcal{F}}_{1,c})^{-1}$ and $R^{\mathcal{F}}_{1,c}$ have no way to know what values of x_i, y_i and x_j, y_j would generate the given c_i and c_j, so they generate x_i, y_i and x_j, y_j uniformly at random, but it is likely that $\mathcal{F}(x_i, y_i) \neq c_i$ and $\mathcal{F}(x_j, y_j) \neq c_j$. If D later sends the query $(\rightarrow, x, y)$ to $S^{\mathcal{F}}_{1,c}$, the latter generates c by $c = \mathcal{F}(x, y)$, then a collision occurs if $c = c_i$ or $c = c_j$ while $(x, y) \neq (x_i, y_i)$ or $(x, y) \neq (x_j, y_j)$. Similarly, when D sends the query $(\leftarrow, K, z_i)$ to $(R^{\mathcal{F}}_{1,c})^{-1}$, the latter has no way to know what value of c_i would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(x_i, y_i)$ (while generating x_i, y_i uniformly at random). If D later sends the query $(\rightarrow, x, y)$ to $S^{\mathcal{F}}_{1,c}$, the latter generates z by $z = \mathcal{F}(K, c)$, where $c = \mathcal{F}(x, y)$, a collision then occurs if $z = z_i$ while $c \neq c_i$.

4. $(S^{\mathcal{F}}_{1,x})^{-1}$: simulator $(S^{\mathcal{F}}_{1,x})^{-1}$ cannot generate any collision because it accepts queries of the format $(\leftarrow, z, c)$ where both c and z are given, so if they exist in $\mathcal{T}^K_x$, no new tuple will be generate and thus no collision.

5. $(S^{\mathcal{F}}_{1,y})^{-1}$: simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ cannot generate collisions in the value of z because it receives it as part of its query $(\leftarrow, z, x)$, so if a given z matches an existing z_i in $\mathcal{T}^K_{1,y}$, no new tuple will be generate and thus no collision. However, when D sends the queries $(\rightarrow, x_i, c_i)$ and $(\leftarrow, K, c_j)$ to $S^{\mathcal{F}}_{1,y}$ and $(R^{\mathcal{F}}_{1,y})^{-1}$, respectively, both $S^{\mathcal{F}}_{1,y}$ and $(R^{\mathcal{F}}_{1,y})^{-1}$ have no way to know what values of y_i and y_j would generate the given c_i and c_j, so they generate y_i and y_j uniformly at random. If D later sends the query $(\leftarrow, z, x)$ to $(S^{\mathcal{F}}_{1,y})^{-1}$, the latter first generates y uniformly at random and then generates c by $c = \mathcal{F}(K, y)$, a collision then occurs if $c = c_i$ or $c = c_j$ while $y \neq y_i$ or $y \neq y_j$.

6. $(S^{\mathcal{F}}_{1,c})^{-1}$: simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ cannot generate collision in the value of c because it receives it as part of its query, so if a given c matches an existing c in $\mathcal{T}^K_c$, no new tuple will be generate and thus no collision. However, when D sends the query $(\leftarrow, K, z_i)$ to $(\mathcal{R}^{\mathcal{F}}_{1,c})^{-1}$, the z_i is given in the query, so there is no way for $(\mathcal{R}^{\mathcal{F}}_{1,c})^{-1}$ to know what value of c_i would generate the given z_i, so it generates c_i by $c_i = \mathcal{F}(x_i, y_i)$, where x_i, y_i are generated uniformly at random. If D later sends $(\leftarrow, c, x)$ to $(S^{\mathcal{F}}_{1,c})^{-1}$, the latter generates z by $z = \mathcal{F}(K, \mathcal{F}(c))$, a collision then occurs if $z = z_i$ while $c \neq c_i$.

Scenarios 1,2 and 3 above are the failure conditions 1(c).i., 1(c).ii. and 1(c).iii., respectively, while scenarios 5 and 6 are the failure conditions 2(c).i.,2(c).ii., respectively (probability of scenario 4 is 0, as discussed above). Failure conditions 1(c).ii. and 1(c).iii. may occur with probability $\leq (q_2 + q_3)^2/2^{m+n}$; in this case, we should consider the collision probability of a *combination* of $x \in \{0,1\}^m$ and $c \in \{0,1\}^n$ (in the case of 1(c).ii.) or $x \in \{0,1\}^m$ and $y \in \{0,1\}^n$ (in the case of 1(c).iii.) since it is the combination what causes the collision not the individual instances of x, y or c, x. Following similar argument, failure condition 1(c).i. occur with probability $\leq (q_2 + q_3)^2/2^{2n}$ as the colliding strings in this case are both of length n-bits, that is, we should consider the collision probability of a combination of $c \in \{0,1\}^n$ and $y \in \{0,1\}^n$. In all the failure conditions of 1(c), the collision occurs by two colliding strings, each consists of two values, where the combination of these two values in each string is what causes the collision; this is discussed above and reflected on their probabilities. On the other hand, in failure conditions 2(c).i. and 2(c).ii., the two colliding strings consist of a single value each ($y \in \{0,1\}^n$ in the case of 2(c).i. and $c \in \{0,1\}^n$ in the case of 2(c).ii.). Thus, both failure conditions 2(c).i. and 2(c).ii. may occur with probability $\leq (q_2 + q_3)/2^{2n}$.

Finally, failure condition 2(d) covers collisions caused by partially matched queries. For example, if a simulator received a 2-string query (x, y), it first searches its corresponding table for a tuple $(x_i, y_i, c_i, z_i, t_i)$ such that $(x, y) = (x_i, y_i)$, here both x and y should match x_i and y_i, respectively. However, if one of these two strings (either x or y) was a match (with x_i or y_i), the simulator will ignore it and proceed to generate a new tuple. This is not an issue if the simulator received a forward query because forward queries generate either c or z, so there will be no collision as along as one of the received strings is distinct (unless one of the other failure conditions is satisfied). On the other hand, this becomes problematic when the simulators receive inverse queries because in this case they will receive c or z as part of the query and this may lead to a collision. We now consider all the inverse simulators $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (S^{\mathcal{F}}_{1,c})^{-1}$ and show how such collisions can occur:

1. $(S^{\mathcal{F}}_{1,x})^{-1}$: when simulator $(S^{\mathcal{F}}_{1,x})^{-1}$ receives the query (z, c), if there is tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_x$, then no new tuple will be created. On the other hand, if $c = c_i$ and $z \neq z_i$, a new tuple will be created, but this will not generate a collision since $z \neq z_i$. However, if it is the other way round, $c \neq c_i$ and $z = z_i$, a new tuple will be created, but this time it will cause a collision since $(c, y) \neq (c_i, y_i)$ while $z = z_i$.

2. $(S^{\mathcal{F}}_{1,y})^{-1}$: fFollowing the same argument, when simulator $(S^{\mathcal{F}}_{1,y})^{-1}$ receives the query (z, x), new tuple will be created even if there is a tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_y$ such that $x \neq x_i$ while $z = z_i$, this will obviously lead to a collision since $(x, c) \neq (x_i, c_i)$ while $z = z_i$.

3. $(S^{\mathcal{F}}_{1,c})^{-1}$: with simulator $(S^{\mathcal{F}}_{1,c})^{-1}$, partial query collision leads to a collision in the c value, but this obviously results in a collision with the z value since, according to $(S^{\mathcal{F}}_{1,c})^{-1}$ definition, $z \leftarrow \mathcal{F}(K, \mathcal{F}(c))$, so as long as there is a collision in c, there will also be a collision in z.

All the scenarios in failure condition 2(d) occur with probably bounded by the birthday attack. In 2(d).i. the collision occurs due to $c \in \{0,1\}^n$, thus the probably is bound by $(q_2 + q_3)/2^n$. On

the other hand, in failure conditions 2(d).ii. and 2(d).iii., the collisions occur due to $x \in \{0,1\}^m$, so the probability is bounded by $(q_2+q_3)^2/2^m$. Note that partial query collisions cannot occur with simulators $R^{\mathcal{F}}_{1,i}$ and $(R^{\mathcal{F}}_{1,i})^{-1}$ (where $i \in \{x,y,c\}$) because these simulators accept a single string (in addition to the fixed key K), so if it matches one of the existing tuple, it will be detected immediately.

Other than the failure conditions above, we prove that collisions in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ cannot occur. As there are two types of tuples in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ (sequenced and singular), there are four possible collision scenarios, namely: collisions between tuples from different sequences, collisions among tuples within the same sequence, collisions among singular tuples, and collisions between sequenced and singular tuples; these are covered by lemmas 4.2, 4.4, 4.6 and 4.7, respectively. These proofs apply for both the modified S simulators in this game, and the modified R simulators in $G(5)$; that is, in $G(5)$ we will modify the simulators $R^{\mathcal{F}}_x, R^{\mathcal{F}}_y, R^{\mathcal{F}}_c$ (and their inverse variants) and introduce failure conditions similar to the ones we introduced in this game, then the distinguishing probability of $G(5)$ will be the success probability of the failure conditions there, other than those failure conditions, the following lemmas prove that $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$ (and their inverse variants) cannot generate collisions.

Lemma 4.2 (Collision freeness among sequences)**.** *For any two sequences, Seq_1 and Seq_2, in a table $\mathcal{T}^K_i$ that is maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}$, $R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, a tuple in Seq_1 cannot collide with another in Seq_2, where $i \in \{x,y,c\}$.*

Proof. Let s_h and s_f index two different sequences, Seq_1 and Seq_2, in the table $\mathcal{T}^K_i, i \in \{x,y.c\}$, consisting of u and v tuples, respectively, and let Seq_1 be rooted by the i-th tuple while Seq_2 be rooted by the j-th tuple:

$$\begin{aligned} Seq_1 &= (x_i^{s_h,0}, y_i^{s_h,0}, c_i^{s_h,0}, z_i^{s_h,0}, t_i^{s_h,0}) \ldots (x_-^{s_h,u}, y_-^{s_h,u}, c_-^{s_h,u}, z_-^{s_h,u}, t_-^{s_h,u}) \\ Seq_2 &= (x_j^{s_f,0}, y_j^{s_f,0}, c_j^{s_f,0}, z_j^{s_f,0}, t_j^{s_f,0}) \ldots (x_-^{s_f,v}, y_-^{s_f,v}, c_-^{s_f,v}, z_-^{s_f,v}, t_-^{s_f,v}) \end{aligned}$$

where $y_i^{s_h,0} = y_j^{s_f,0} = IV$ and $h \neq f$. A collision between tuple $(x_a, y_a, c_a, z_a, t_a)$ in Seq_1 and tuple $(x_b, y_b, c_b, z_b, t_b)$ in Seq_2 occurs when $z_a = z_b$ while $(x_a, y_a, c_a) \neq (x_b, y_b, c_b)$. Thus, we need to show that whenever a sequenced tuple is created, it cannot collide with any other existing sequenced tuple. Here our discussion is based on the assumption that the z value of a tuple is being generated by the simulators, but in simulators $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$, the z value is given to the simulators as part of the queries. These simulators, however, will only generate singular tuples while here we are only concerned with sequenced tuples. In fact, the only simulators that will generate sequences are $S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$, so it suffices to investigate how these simulators generate their z values because if there is a collision, they are the only ones that could have cause it. The easiest way to do this is to observe the input that the simulators use to generate their z values and prove that z will be unique as long as these inputs are unique (generated in a collision-free manner).

Simulator $S^{\mathcal{F}}_{1,x}$ generates its z by $\mathcal{F}(\mathcal{F}(c), y)$, simulator $R^{\mathcal{F}}_{1,y}$ generates its z by $\mathcal{F}(T||x, c)$, and simulator $S^{\mathcal{F}}_{1,c}$ generates its z by $\mathcal{F}(K, T||x)$, where T is a sequence of x's. We now show that the inputs $(\mathcal{F}(c), y), (T||x, c), (K, T||x)$ are prepared in a collision-free manner (i.e., they are unique) which will immediately imply that the z values they will generate are also collision-free since the latter is generated by a $\mathcal{RO}$.

- $S^{\mathcal{F}}_{1,x}(\rightarrow, c, y)$: in this case, the z value is created by the query $(\mathcal{F}(c), y)$ to the random oracle $\mathcal{F}$, so we show that both c and y are unique. It is clear that the y value here cannot collide with any existing y value because the simulator receives y as part of its query. If there is an existing tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}^K_x$ such that $y_i = y$, then $S^{\mathcal{F}}_{1,x}$ will only create

a new tuple if $c \neq c_i$, otherwise if $c = c_i$, then $(c, y) = (c_i, y_i)$ and a new tuple will not be created. This also includes the case when $y = IV$, where a new tuple will only be created if $c \neq c_i$. This mean that the $S^{\mathcal{F}}_{1,x}$ simulator guarantees a total collision freeness, which implies that $\mathcal{F}(\mathcal{F}(c), y) \neq \mathcal{F}(\mathcal{F}(c_i), y_i)$ will hold for any $(c_i, y_i) \in \mathcal{T}^K_x$, which immediately implies $z_i \neq z_j$.

- $R^{\mathcal{F}}_{1,y}(\rightarrow, K, y)$: this simulator creates its z value by querying $\mathcal{F}$ with $(T||x, c)$, where T is a sequence of the x's of the preceding tuples in the sequence to which the new query belongs. It is easy to see that T is unique for a particular z because this sequence of x's will only occur for that particular query. On the other hand, c here is being generated by $\mathcal{F}(K, y)$, which implies that as long as $y \neq y_i$ for some $y_i \in \mathcal{T}^K_y$, then $\mathcal{F}(K, y_i) \neq \mathcal{F}(K, y_j)$ holds, which, in turn, implies $c_i \neq c_j$, and this will always be the case since y here is received as part of the query and if it matches any $y_i \in \mathcal{T}^K_y$, no new tuple will be created.

- $S^{\mathcal{F}}_{1,c}(\rightarrow, x, y)$: in this simulator, the z value is created by querying $(K, T||x)$ to the random oracle $\mathcal{F}$, where K is a fixed key and T is a sequence of x's. In this case, the collision freeness of z solely depends on the value T, which, as discussed above, is unique for any particular query.

Finally, it is easy to see that these results will also generalise to cases when there are multiple sequences in $\mathcal{T}^K_i$ where $i \in \{x, y, c\}$. □

Corollary 4.3 (Prefix collision freeness among sequences)**.** *If any of the tables $\mathcal{T}^K_i$ maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}, R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, where $i \in \{x, y, c\}$, contains two different sequences Seq_1 and Seq_2, then any tuple in Seq_1 cannot be a prefix of any tuple in Seq_2, and vice versa.*

Proof. The proof follows from lemma 4.2. A tuple $(x_a^{s_h,p}, y_a^{s_h,p}, c_a^{s_h,p}, z_a^{s_h,p}, t_a^{s_h,p})$ in Seq_1 cannot be a prefix of another tuple $(x_b^{s_f,q}, y_b^{s_f,q}, c_b^{s_f,q}, z_b^{s_f,q}, t_b^{s_f,q})$ in Seq_2, that is $z_a^{s_h,p}$ cannot equal to $y_b^{s_f,q}$, because if that was the case, then this is merely a collision between $z_a^{s_h,p}$ and $z_{\_}^{s_f,q-1}$ since $y_b^{s_f,q} = z_{\_}^{s_f,q-1}$, and as shown in lemma 4.2, a collision cannot occur in this case. Clearly, the other way round also holds; no tuple from Seq_2 may be a prefix to another in Seq_1. Also, a sequence cannot be a prefix of itself. That is, given a sequence Seq_3 rooted at the i-th tuple and has a tail at the j-th tuple, then $y_i \neq z_j$ holds since $y_i = IV$. However, with probability $1/2^n$ (where $n = |z_j|$), it might be the case that $z_j = IV$, but this is covered in failure conditions 1(a) and 2(a), which also covers prefix collisions within a single sequenced tuple. □

Lemma 4.4 (Collision freeness within a single sequence)**.** *If any of the tables $\mathcal{T}^K_i$ maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}, R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, where $i \in \{x, y, c\}$, contains a sequence Seq_π, then any tuple in that sequence cannot collide with any other tuple in the same sequence.*

Proof. Recall that the only simulators generating sequences are $S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$. Let $Seq_\pi = (x_a^{s_l,0}, y_a^{s_l,0} = IV, c_a^{s_l,0}, z_a^{s_l,0}, t_a^{s_l,0}) \dots (x_{\_}^{s_l,n}, y_{\_}^{s_l,n}, c_{\_}^{s_l,n}, z_{\_}^{s_l,n}, t_{\_}^{s_l,n})$ be a sequence containing $n > 1$ tuples and rooted by the a-th tuple. An internal collision in the sequence s_l implies that there exists $z_{\_}^{s_l,i} = z_{\_}^{s_l,j}$, where $i \neq j$ and $i, j \in \{0, \dots, n\}$. In $R^{\mathcal{F}}_{1,y}$ and $S^{\mathcal{F}}_{1,c}$, this happens if the following equality holds:

$$x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \dots ||x_{\_}^{s_l,i-1}||x_{\_}^{s_l,i} = x_{\_}^{s_l,0}||x_{\_}^{s_l,1}|| \dots ||x_{\_}^{s_l,j-1}||x_{\_}^{s_l,j}$$

Since z is being generated by the random oracle $\mathcal{F}$, this is only possible if $i = j$. On the other hand, in $S^{\mathcal{F}}_{1,x}$, an internal collision happens between two tuples $(x_p, y_p, c_p, z_p, t_p)$ and $(x_q, y_q, c_q, z_q, t_q)$ belonging to the same sequence if $z_p = z_q$. This can happen only if $(c_p, y_p) = (c_q, y_q)$, which is not possible since both c and y are given to $S^{\mathcal{F}}_{1,x}$ as part of the query and would

only cause a new tuple to be created if there is no existing tuple matching the queried c and y. □

Corollary 4.5 (Ancestors and descendants of sequenced tuples)**.** *In any of the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, a single tuple cannot be part of more than one sequence. More generally, a single tuple in $\mathcal{T}_i^K, i \in \{x, y, c\}$ cannot have more than one descendant tuple and more than one parent tuple.*

Proof. The proof follows from lemmas 4.2 and 4.4. Let the i-th tuple indexed by s_l, p be the parent of the tuple indexed by $s_l, p+1$ of the sequence l, that is $z_i^{s_l,p} = y_{\_}^{s_l,p+1}$. The only way the tuple s_l, p can have another descendant s_l, k is when $z_i^{s_l,p} = y_{\_}^{s_i,k}$, which implies that $y_{\_}^{s_i,p+1} = y_{\_}^{s_i,k}$, but this cannot happen because collisions and prefixes cannot occur as shown in lemmas 4.2 and 4.4. Similarly, and following the same argument, tuple s_l, p with parent $s_l, p-1$ cannot have another parent s_l, k' because this implies $z_{\_}^{s_l,p-1} = z_{\_}^{s_l,k'} = y_i^{s_l,p}$, which cannot occur. □

Lemma 4.6 (Collision freeness among singular tuples)**.** *In any of the tables $\mathcal{T}_i^K$ maintained by the simulators $S_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, R_{1,i}^{\mathcal{F}}, (R_{1,i}^{\mathcal{F}})^{-1}$, where $i \in \{x, y, c\}$, collisions cannot occur between two singular tuples within the same table.*

Proof. A collision in this context means there are two tuples $(x_i, y_i, c_i, z_i, t_i = \perp)$ and $(x_j, y_j, c_j, z_j, t_j = \perp)$, such that $z_i = z_j$ while $(x_i, y_i, c_i) \neq (x_j, y_j, c_j)$. However, since z is always being generated by the random oracle $\mathcal{F}$ (unless given by D), $z_i = z_j$ will only hold if the inputs given to $\mathcal{F}$ to generate z_i and z_j are identical. In lemma 4.2 we considered how sequenced tuples are being generated by $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}$, in this proof, we need to consider the other simulators which generate singular tuples (in addition to $S_{1,x}^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}$ since these can also generate singular tuples), these are:

$$(S_{1,x}^{\mathcal{F}})^{-1}, S_{1,y}^{\mathcal{F}}, (S_{1,y}^{\mathcal{F}})^{-1}, (S_{1,c}^{\mathcal{F}})^{-1}, (R_{1,y}^{\mathcal{F}})^{-1}, R_{1,x}^{\mathcal{F}}, (R_{1,x}^{\mathcal{F}})^{-1}, R_{1,c}^{\mathcal{F}}, (R_{1,c}^{\mathcal{F}})^{-1}$$

In all these cases, unless given by D, the value of z is generated based on c, so it only suffices to prove that in each case, the way c is being generated is collision-free to imply that z will be collision-free. Precisely, z_i of a singular tuple can either by generated by $z_i = \mathcal{F}(X, c_i)$ or $z = \mathcal{F}(X, \mathcal{F}(c_i))$, where $X \in \{K, x_i, y_i\}$. Therefore, we now have two possible inputs to z, making four possible collision scenarios:

- $z_i = \mathcal{F}(X, c_i), z_j = \mathcal{F}(X, c_j)$: the only simulator that generates $z_i = \mathcal{F}(X, c_i)$ is $R_{1,x}^{\mathcal{F}}$, so we show that $R_{1,x}^{\mathcal{F}}$ cannot generate $c_i = c_j$ while $i \neq j$. It is easy to see that this is always the case since $R_{1,x}^{\mathcal{F}}$ uses $\mathcal{F}$ to generate $c_i = \mathcal{F}(K, x_i), c_j = \mathcal{F}(K, x_j)$, so as long as $(K, x_i) \neq (K, x_j)$, then $c_i \neq c_j$ will hold, which is always the case since $R_{1,x}^{\mathcal{F}}$ receives x_i in the query and will only use it to create a new tuple if there is no $x_j \in \mathcal{T}_x^K$ such that $x_i = x_j$.

- $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), z_j = \mathcal{F}(X, \mathcal{F}(c_j))$: apart from $R_{1,x}^{\mathcal{F}}$, all other simulators that generate singular tuples generate z as $\mathcal{F}(X, \mathcal{F}(c_i))$, so we only need to show that $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ will always hold if $c_i \neq c_j$. In all simulators generating singular tuples, c_i is given in the query, so upon receiving a query (X, c_i), the simulators first check if there is a tuple $(x_j, y_j, c_j, z_j, t_j) \in \mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ such that $c_i = c_j$, if it does, they do not create a new tuple. Thus, as long as $c_i \notin \mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, then $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ follows implying $z_i \neq z_j$. An even simpler argument is that $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ trivially holds as long as $c_i \neq c_j$ because $\mathcal{F}$ is a random oracle.

- $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), z_j = \mathcal{F}(X, c_j)$ or $z_i = \mathcal{F}(X, c_i), z_j = \mathcal{F}(X, \mathcal{F}(c_j))$: here we only need to show that $\mathcal{F}(c_i) \neq c_j$, where $i \neq j$, to imply $\mathcal{F}(X, \mathcal{F}(c_i)) \neq \mathcal{F}(X, c_j)$. We know that c_j in $\mathcal{F}(X, c_j)$ is being generated by $\mathcal{F}(x_j, y_j)$ or $\mathcal{F}(K, x_j)$ or $\mathcal{F}(K, y_j)$. Thus, we have $\mathcal{F}(c_i) = \mathcal{F}(x_j, y_j)$ or $\mathcal{F}(c_i) = \mathcal{F}(K, x_j)$ or $\mathcal{F}(c_i) = \mathcal{F}(K, y_j)$, none of which can hold since $\{0,1\}^n \neq [\{0,1\}^m || \{0,1\}^n], \{0,1\}^n \neq [\{0,1\}^k || \{0,1\}^m], \{0,1\}^n \neq [\{0,1\}^k || \{0,1\}^n]$, respectively.

It remains to discuss the case when z is given in the query, which can happen only with $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$, but it is easy to see that $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$ will not generate collisions because, upon a new query, they will first check $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$, respectively, for any tuple with a similar z and would only create a new tuple if no such z exists. This covers all possible collision scenarios among singular tuples. □

Lemma 4.7 (Collision freeness between singular and sequenced tuples)**.** *If any of the tables $\mathcal{T}^K_i$ maintained by the simulators $S^{\mathcal{F}}_{1,i}, (S^{\mathcal{F}}_{1,i})^{-1}, R^{\mathcal{F}}_{1,i}, (R^{\mathcal{F}}_{1,i})^{-1}$, where $i \in \{x, y, c\}$, contains a sequence Seq_π, then no singular tuple within the same table may collide with any tuple in Seq_π.*

Proof. Here we have two cases: either the newly generated tuple $(x_i, y_i, c_i, z_i, t_i)$ is singular while colliding with an existing sequenced tuple $(x_j, y_j, c_j, z_j, t_j)$, or the other way round (collision implies $z_i = z_j$). Either way, according to the definition of simulators $S^{\mathcal{F}}_{1,k}, R^{\mathcal{F}}_{1,k}, k \in \{x, y, c\}$, regardless of whether it was singular or sequenced tuple, z will always be generated by $\mathcal{F}$ (unless it is given by D). Let $(x_i, y_i, c_i, z_i, t_i = \perp)$ be a singular tuple while $(x_j^{s_l,p}, y_j^{s_l,p}, c_j^{s_l,p}, z_j^{s_l,p}, t_j^{s_l,p} = 1)$ be a sequenced tuple, with $p \geq 1$. In this case, $z_i = \mathcal{F}(X, c_i)$ or $z_i = \mathcal{F}(X, \mathcal{F}(c_i))$, where $X \in \{K, x_i, y_i\}$, while $z_j = \mathcal{F}(K, x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p})$ when generated by $S^{\mathcal{F}}_{1,c}$ or $z_j = \mathcal{F}(c_j, x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p})$ when generated by $R^{\mathcal{F}}_{1,y}$ or $z_j = \mathcal{F}(c_j, y_j)$ when generated by $S^{\mathcal{F}}_{1,x}$. When z_j is generated by $S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,y}$, unless $c_i = x_{\_}^{s_l,0} || x_{\_}^{s_l,1} || \ldots || x_{\_}^{s_l,p-1} || x_{\_}^{s_l,p}$, $z_i \neq z_j$ will always hold. Similarly, when z_j is generated by $S^{\mathcal{F}}_{1,x}$, $z_j \neq z_i$ will hold as long as $(c_j, y_i) \neq (c_i, y_i)$ also holds, which is always the case as such collision would have been detected by $S^{\mathcal{F}}_{1,x}$ before creating the j-th or i-th tuple (whichever queried last). It is also possible that the sequence l contains only one tuple, that is, only the root tuple $(x_j^{s_l,0}, y_j^{s_l,0} = IV, c_j^{s_l,0}, z_j^{s_l,0}, t_j^{s_l,0})$, in which case a collision occurs if $z_i = z_j^{s_l,0}$, which is not possible as long as $c_i \neq c_j^{s_l,0}$ since both z_i, z_j are generated by $\mathcal{F}$ and c is always involved in their generation process. According to the definition of the simulators $S^{\mathcal{F}}_{1,k}, R^{\mathcal{F}}_{1,k}, k \in \{x, y, c\}$, unless c is given by D, it will be generated by $\mathcal{F}$. If c is sequenced it is generated by $\mathcal{F}(x, y)$ or $\mathcal{F}(K, y)$, otherwise if c is singular it is generated by $\mathcal{F}(K, y)$ or $\mathcal{F}(K, x)$ or $\mathcal{F}(x, y)$. It is easy to see that the fact that $y_j^{s_l,0} = IV$ while $y_i \neq IV$ always holds (otherwise y_i would not be singular) implies that the following also hold (and thus any $c_j \neq c_i$).

$$\Big[\mathcal{F}(x, y) \neq \mathcal{F}(K, y)\Big], \Big[\mathcal{F}(x, y) \neq \mathcal{F}(K, x)\Big], \Big[\mathcal{F}(K, y) \neq \mathcal{F}(K, x)\Big]$$

To complete the proof, it remains to investigate cases when c is given in a query. In all simulators generating singular tuples, when c is given by D, c is not directly used to generate z, rather, $z_i = \mathcal{F}(X, \mathcal{F}(c_i)), X \in \{x_i, y_i\}$, so a collision between a singular tuple and a sequenced one implies $\mathcal{F}(c_i) = c_j^{s_l,0}$ should hold[3], which, for the case of $S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_{1,y}$, can be rewritten as $\mathcal{F}(c_i) = \mathcal{F}(x_j^{s_l,p} || y_j^{s_l,p})$ and $\mathcal{F}(c_i) = \mathcal{F}(K || y_j^{s_l,p})$, but clearly $c_i \neq x_j^{s_l,p} || y_j^{s_l,p}$ and $c_i \neq K || y_j^{s_l,p}$, since $\{0,1\}^n \neq \{0,1\}^m || \{0,1\}^n$ and $\{0,1\}^n \neq \{0,1\}^k || \{0,1\}^n$, respectively. For the case when c_j is generated by $S^{\mathcal{F}}_{1,x}$, $\mathcal{F}(c_i) \neq \mathcal{F}(c_j)$ still holds because $\mathcal{F}(c_i) \neq \mathcal{F}(\mathcal{F}(c_j), y_j)$

[3]Recall that the only simulators generating sequences are $S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ and that $R^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ generate c by $\mathcal{F}(K, y), \mathcal{F}(x, y)$, respectively, while $S^{\mathcal{F}}_{1,x}$ receives c as part of its queries.

since $\{0,1\}^n \neq \{0,1\}^n || \{0,1\}^n$. Finally, it is trivial to see that $(S^{\mathcal{F}}_{1,x})^{-1}, (S^{\mathcal{F}}_{1,y})^{-1}, R^{\mathcal{F}}_{1,c})^{-1}$ will not generate collisions, even though they accepts z from D, because if a queried z already exists in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ no new (colliding) tuple will be created. □

Finally, the probability of this game is the sum of the probabilities of the failure conditions of $S^{\mathcal{F}}_{1,x}, S^{\mathcal{F}}_{1,y}, S^{\mathcal{F}}_{1,c}$ (and their inverse variants):

$$\begin{aligned}
P^x_3 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_x} = 1] &\leq \big(2^n((q_2+q_3)+(q_2+q_3)^2)+(q_2+q_3)\big)/2^{2n} \\
P^y_3 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_y} = 1] &\leq \big((q_2+q_3)(2^{m+1}(q_2+q_3)+2^{m+1}+2^n(q_2+q_3)+1)\big)/2^{m+n} \\
P^c_3 = \Pr[D^{F^{\mathcal{F}}_1, S^{\mathcal{F}}_{1,c}, R^{\mathcal{F}}_c} = 1] &\leq \big((q_2+q_3)(3\cdot 2^m+2^{m+1}(q_2+q_3)+2^n(q_2+q_3)+1))\big)/2^{m+n}
\end{aligned}$$

Game 4. Similar to $G(3)$, in this game we introduce slightly modified replicas of the simulators $R^{\mathcal{F}}_x, R^{\mathcal{F}}_y, R^{\mathcal{F}}_c$, still with oracle access to $\mathcal{F}$. The distinguisher D will now interact with the modified simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$ which, upon receiving a query, create the new tuple (x,y,c,z,t) but explicitly abort if any of the following failure conditions is satisfied:

1. On forward queries $R^{\mathcal{F}}_{1,x}(\rightarrow,K,x), R^{\mathcal{F}}_{1,y}(\rightarrow,K,y), R^{\mathcal{F}}_{1,c}(\rightarrow,K,c)$, the simulators create the new tuple (x,y,c,z,t), but the following collisions occur:
   - 1(a). Fixed point: in $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$, it is the case that $z = IV$, and in $R^{\mathcal{F}}_{1,y}$, it is also the case that $z = y$.
   - 1(b). Prefix collision: in $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$, it is the case that $z = y_j$ for some $y_j \in \mathcal{T}^K_x(\hat{y}) \cup \mathcal{T}^K_y(\hat{y}) \cup \mathcal{T}^K_c(\hat{y})$.
   - 1(c). Internal collision: there is a tuple $(x_i, y_i, c_i, z_i, t_i)$ in $\mathcal{T}^K_x, \mathcal{T}^K_y, \mathcal{T}^K_c$ such that:
     - i. when $R^{\mathcal{F}}_{1,x}$ receives $(\rightarrow, K, x)$, the following hold: $c_i = c \wedge x_i \neq x$.
     - ii. when $R^{\mathcal{F}}_{1,y}$ receives $(\rightarrow, K, y)$, the following hold: $c_i = c \wedge y_i \neq y$.
2. On inverse queries $(R^{\mathcal{F}}_{1,x})^{-1}(\leftarrow,K,c), (R^{\mathcal{F}}_{1,y})^{-1}(\leftarrow,K,c), (R^{\mathcal{F}}_{1,c})^{-1}(\leftarrow,K,z)$, the simulators create the new tuple (x,y,c,z,t), but the following collisions occur:
   - 2(a). Fixed point: in $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}$, it is the case that $z = IV$.
   - 2(b). Prefix collision: in $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$, it is the case that $z = y_j$ for some $y_j \in \mathcal{T}^K_x(\hat{y}) \cup \mathcal{T}^K_y(\hat{y})$.
   - 2(c). Internal collision: we show that $(R^{\mathcal{F}}_{1,x})^{-1}, (R^{\mathcal{F}}_{1,y})^{-1}, (R^{\mathcal{F}}_{1,c})^{-1}$ cannot generate collisions, see below.

As illustrated in $G(3)$, failure conditions 1(a) and 2(a) occur with probability $(q_2+q_3)/2^n$ each. Simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,y}, R^{\mathcal{F}}_{1,c}$ are susceptible to the first fixed point type (when $z = IV$), while only simulator $R^{\mathcal{F}}_{1,y}$ is susceptible to the second fixed point type (when $z = y_i$), in the latter case simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,c}$ are not susceptible to the second fixed point type because they generate their y value uniformly at random excluding all existing values of z and y in tables $\mathcal{T}^K_x, \mathcal{T}^K_c$. Thus, failure condition 1(a) occurs for simulators $R^{\mathcal{F}}_{1,x}, R^{\mathcal{F}}_{1,c}$ with probability $(q_2+q_3)/2^n$, while it occurs for simulator $R^{\mathcal{F}}_{1,y}$ with probability $2(q_2+q_3)^2/2^n$. Similar analysis apply for failure condition 2(a).

Following the same argument in $G(3)$, both failure conditions 1(b) and 2(b) are at most the birthday bound $(q_2+q_3)^2/2^n$ and are applicable to all simulators in this game. In particular, all simulators except $(R^{\mathcal{F}}_{1,c})^{-1}$ generate their z values by querying a $\mathcal{RO}$, meaning that they

cannot force the $\mathcal{RO}$ to exclude the existing y values in the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ from the newly generated z value. In the case of $(R_{1,c}^{\mathcal{F}})^{-1}$, z is given as part of the query and the simulator only checks whether there is a tuple $(x_i, y_i, c_i, z_i, t_i) \in \mathcal{T}_c^K$ such that $z = z_i$ before creating a new tuple, it, however, does not check for $z = y_i$, so this prefix collision can still happen with probability $(q_2 + q_3)^2/2^n$. We now discuss in details failure conditions 1(c) and 2(c).

1. $R_{1,x}^{\mathcal{F}}$: When D sends the query $(\rightarrow, c, y)$ to $S_{1,x}^{\mathcal{F}}$, the latter has no way to know which x generated the given c, so it generates x uniformly at random, but it is most likely that $\mathcal{F}(K, x) \neq c$. When D later sends the query $(\rightarrow, K, x_i)$ to $R_{1,x}^{\mathcal{F}}$, the latter generates $c_i = (K, x_i)$. A collision occurs if $c_i = c \wedge x_i \neq x$.

2. $R_{1,y}^{\mathcal{F}}$: When D sends the query $(\rightarrow, x, c)$ to $S_{1,y}^{\mathcal{F}}$, the latter has no way to know which y generated the given c, so it generates y uniformly at random, but it is most likely that $\mathcal{F}(K, x) \neq c$. When D later sends the query $(\rightarrow, K, y_i)$ to $R_{1,y}^{\mathcal{F}}$, the latter generates $c_i = (K, x_i)$. A collision occurs if $c_i = c \wedge y_i \neq y$.

3. Simulators $R_{1,c}^{\mathcal{F}}$, $(R_{1,x}^{\mathcal{F}})^{-1}$, $(R_{1,y}^{\mathcal{F}})^{-1}$, $(R_{1,c}^{\mathcal{F}})^{-1}$ do not generate any collision because the queries made to them by D contain either c or z values, which if they match any of the existing c or z values in $\mathcal{T}_x^K$ or $\mathcal{T}_y^K$ or $\mathcal{T}_c^K$, no new tuple is generate and thus no collision.

Scenarios 1 and 2 are are failure conditions 1(c).i. and 2(c).ii., respectively. Failure condition 1(c).i may occur with probability $\leq (q_2 + q_3)/2^m$ since the probably is taken over range size of 2^m (i.e., $x \in \{0,1\}^m$). In this case, x is generated uniformly at random and is assumed to have generated the c value that was sent the query $(\rightarrow, c, y)$, but when later D sends a query with x_i (i.e. $(\rightarrow, K, x_i)$) and it turned out that x_i is the value that really generates the previously sent c when given to a $\mathcal{RO}$, then x and x_i collide at c, this happens with probability $1/2^m$ since there is only one x value generating a particular c; after $q_2 + q_3$ queries, the probability is $(q_2 + q_3)/2^m$. Following similar argument, scenario 2(c).i. may occur with probability $\leq (q_2 + q_3)/2^n$ since the collision occurs between strings of size n-bit, (i.e., $y \in \{0,1\}^n$). Finally, failure condition 2(c) occur with probability 0, as discussed in scenario 3 above. Thus, the final probability of $G(4)$ is:

$$\begin{aligned}
P_4^x = \Pr[D^{F_1^{\mathcal{F}}, S_{1,x}^{\mathcal{F}}, R_{1,x}^{\mathcal{F}}} = 1] &\leq \left(2(q_2+q_3) + 2(q_2+q_3)^2\right)/2^n + (q_2+q_3)/2^m \\
P_4^y = \Pr[D^{F_1^{\mathcal{F}}, S_{1,y}^{\mathcal{F}}, R_{1,y}^{\mathcal{F}}} = 1] &\leq \left(3(q_2+q_3) + 3(q_2+q_3)^2\right)/2^n \\
P_4^c = \Pr[D^{F_1^{\mathcal{F}}, S_{1,c}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}}} = 1] &\leq \left(q_2+q_3+2(q_2+q_3)^2\right)/2^n
\end{aligned}$$

Game 5. In this game, we modify the relay algorithms $F_{1,x}^{\mathcal{F}}, F_{1,y}^{\mathcal{F}}, F_{1,c}^{\mathcal{F}}$ to make them dependant on $(S_{1,x}^{\mathcal{F}}, R_{1,x}^{\mathcal{F}}), (S_{1,y}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}}), (S_{1,c}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}})$ instead of $\mathcal{F}$, and thus simulating the constructions $\hat{C}_x^{\mathcal{H}_x, \mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y, \mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c, \mathcal{G}_c}$, respectively (as defined in figure 1). We prove that $F_{1,x}^{\mathcal{F}}, F_{1,y}^{\mathcal{F}}, F_{1,c}^{\mathcal{F}}$ and $F_{2,x}^{S_{1,x}, R_{1,x}}, F_{2,y}^{S_{1,y}, R_{1,y}}, F_{2,c}^{S_{1,c}, R_{1,c}}$ (the modified relay algorithms) behave consistently as long as the sequenced tuples in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ are preserved.

Lemma 4.8 (Indistinguishability of G(5))**.** *The modified relay algorithms $F_{2,x}^{S_{1,x}, R_{1,x}}, F_{2,y}^{S_{1,y}, R_{1,y}}$, $F_{2,c}^{S_{1,c}, R_{1,c}}$, with access to the simulators $(S_{1,x}^{\mathcal{F}}, R_{1,x}^{\mathcal{F}}), (S_{1,y}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}})$, $(S_{1,c}^{\mathcal{F}}, R_{1,c}^{\mathcal{F}})$, respectively, are either indistinguishable or behave consistently with $F_{1,x}^{\mathcal{F}}, F_{1,y}^{\mathcal{F}}$, $F_{1,c}^{\mathcal{F}}$, relay algorithms with oracle access to a random oracle $\mathcal{F}$.*

Proof. Let $X = x_1 || x_2 || \ldots || x_n$ be a message consisting of n m-bit blocks, and let K be a fixed k-bit key. When X is given as a query, $F_{1,c}^{\mathcal{F}}$ processes it as a whole by sending it to

the random oracle $\mathcal{F}$, that is $F_{1,i}^{\mathcal{F}}(K,X) = \mathcal{F}(K||x_1||x_2||\ldots||x_n)$ and then $z \in \{0,1\}^n$ is obtained, where $K \in \{0,1\}^k$ is a key and $i \in \{x,y,c\}$. First, we prove that $F_{1,x}^{\mathcal{F}}(K,X)$ is indistinguishable from $F_{2,x}^{S_{1,x},R_{1,x}}(K,X)$. Indistinguishability here means that the distinguisher D cannot distinguish between responses from $F_{1,x}^{\mathcal{F}}$ and $F_{2,x}^{S_{1,x},R_{1,x}}$ (except with negligible probability), but not necessarily require the two responses to be identical for similar queries. In fact, $F_{1,x}^{\mathcal{F}}(K,X) = F_{2,x}^{S_{1,x},R_{1,x}}(K,X)$ never holds. To see why, suppose $X = x_1 \in \{0,1\}^m$ and $K \in \{0,1\}^k$, now $F_{1,x}^{\mathcal{F}}(K,x_1) = \mathcal{F}(K||x_1) = v_1$ while $F_{2,x}^{S_{1,x},R_{1,x}}(K,x_1) = S_{1,x}^{\mathcal{F}}(Y, R_{1,x}^{\mathcal{F}}(K,x_1)) = \mathcal{F}(Y,\mathcal{F}(K,x_1)) = v_2$ for some $Y \in \{0,1,\}^n$. Clearly, $v_1 \neq v_2$ always holds since

$$\Big[\mathcal{F}(K||x_1) = \mathcal{F}(\{0,1\}^{k+m})\Big] \neq \Big[\mathcal{F}(K||\mathcal{F}(y_1||x_1)) = \mathcal{F}(\{0,1\}^{k+n})\Big]$$

This will also apply when $X = x_1, x_2, \ldots, x_n$. Furthermore, in $F_{2,x}^{S_{1,x},R_{1,x}}$, getting every input block x_i preprocessed by $R_{1,x}^{\mathcal{F}}$ thwarts other distinguishing attacks. Next, we prove $F_{1,c}^{\mathcal{F}}(K,X) = F_{2,c}^{S_{1,c},R_{1,c}}(K,X)$, the proof of $F_{1,y}^{\mathcal{F}}(K,X) = F_{2,y}^{S_{1,y},R_{1,y}}(K,X)$ is similar. Unlike $F_{1,c}^{\mathcal{F}}$, $F_{2,c}^{S_{1,c},R_{1,c}}$ processes an incoming query by first partitioning it into blocks and then processes each block separately through $S_{1,c}^{\mathcal{F}}$ and $R_{1,c}^{\mathcal{F}}$. Formally, when $F_{2,c}^{S_{1,c},R_{1,c}}$ receives the query (K,X), it beings by dividing X into $x_1, x_2, \ldots, x_n$ and then querying $S_{1,c}^{\mathcal{F}}(IV, x_1)$. Once $S_{1,c}^{\mathcal{F}}$ receives this query it immediately creates a sequence in $\mathcal{T}_c^K$ rooted with the tuple $(x_i = x_1, y_i = IV, c_i, z_i, t_i = 0)$ where c_i and z_i are obtained based on the definition of $S_{1,c}^{\mathcal{F}}$. The simulator $S_{1,c}^{\mathcal{F}}$ will then return c_i to $F_{2,c}^{S_{1,c},R_{1,c}}$ which will immediately send it to $R_{1,c}^{\mathcal{F}}(K,c_i)$ and eventually gets z_i. At this stage $F_{2,c}^{S_{1,c},R_{1,c}}$ has completed processing the first block x_1, so it proceeds to process the second block $S_{1,c}^{\mathcal{F}}(z_i, x_2)$. Once $S_{1,c}^{\mathcal{F}}$ receives this new query, it detects it as a sequenced tuple and links the new tuple $(x_j = x_2, y = z_i, c_j, z_j, t_j = 1)$ to the root tuple (the i-th tuple). Note that z_j is not created randomly, instead $S_{1,c}^{\mathcal{F}}$ queries the random oracle $\mathcal{F}(K, x_1||x_2)$ to obtain this value. The process continues until reaching x_n which will create the tuple $(x_\_ = x_n, y_\_, c_\_, z_\_, t_\_ = 1)$ where $z_\_ = \mathcal{F}(K, x_1||x_2||\ldots||x_n) = F_{1,c}^{\mathcal{F}}(K,X)$. □

It follows then that this game is a syntactical rewrite of the pervious game and the view of D will not change when it interacts with ($F_{1,x}^{\mathcal{F}}$ or $F_{2,x}^{S_{1,x},R_{1,x}}$) and ($F_{1,y}^{\mathcal{F}}$ or $F_{2,y}^{S_{1,y},R_{1,y}}$) and ($F_{1,c}^{\mathcal{F}}$ or $F_{2,c}^{S_{1,c},R_{1,c}}$), except that we now have to account for queries $q_1 \in \{\{0,1\}^m\}^*$ since these will update the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. Previously, when calculating the probabilities we only considered queries q_2 and q_3 because they were the only queries that will access and interact with the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. However, accounting for q_1 is slightly less straightforward than q_2 and q_3 since q_1 have a variable length. Let L denote the maximum length of q_1 (we assume that L is divisible by m). What we are concerned about here is how many times a single q_1 query accesses and probably updates $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ because this is what affects the probability, and we obtain this number by L/m, where m is the length of a single block (recall that a query q_1 will be partitioned into several m-bit blocks which will then be processed sequentially in order by $F_{2,x}^{S_{1,x},R_{1,x}}, F_{2,y}^{S_{1,y},R_{1,y}}, F_{2,c}^{S_{1,c},R_{1,c}}$). Additionally, in this game $F_{2,x}^{S_{1,x},R_{1,x}}, F_{2,y}^{S_{1,y},R_{1,y}}, F_{2,c}^{S_{1,c},R_{1,c}}$ make extra finalising calls to $R_{1,x}, R_{1,y}, R_{1,c}$, and thus the probability of collisions in these calls need to be accounted for[4], which is implicit with the q_1 queries. Therefore, the final probability of this game is the sum of the failure conditions in $G(3)$ and $G(4)$ given the additional q_1 queries.

[4] A subtle technical issue is when $F_{2,x}^{S_{1,x},R_{1,x}}$ calls $R_{1,x}$ to finalise an n-bit string ($R_{1,x}$ originally handles m-bit strings). To resolve this problem, in section 2 we propose padding the n-bits by 0 $m-n$ bits, which will not affect the proof, and have a negligible effect on the running time.

$$
\begin{aligned}
P_5^x &= \Pr[D^{F_2^{\mathcal{F}},S_{1,x}^{\mathcal{F}},R_{1,x}^{\mathcal{F}}} = 1] \\
&\leq \left(2^n(3(q_1 \cdot L/m) + 5(q_1 \cdot L/m)^2) + (q_1 \cdot L/m)\right)/2^{2n} + \left((q_1 \cdot L/m)\right)/2^m \\
P_5^y &= \Pr[D^{F_2^{\mathcal{F}},S_{1,y}^{\mathcal{F}},R_{1,y}^{\mathcal{F}}} = 1] \\
&\leq \left(5(q_1 \cdot L/m) + 5(q_1 \cdot L/m)^2\right)/2^n + \left(2^2(q_1 \cdot L/m)^2 + (q_1 \cdot L/m)\right)/2^{m+n} \\
P_5^c &= \Pr[D^{F_2^{\mathcal{F}},S_{1,c}^{\mathcal{F}},R_{1,c}^{\mathcal{F}}} = 1] \\
&\leq \left((q_1 \cdot L/m)(3 \cdot 2^m + 2^{m+1}(q_1 \cdot L/m) + 2^n(q_1 \cdot L/m) + 1))\right)/2^{m+n} \\
&+ \left((q_1 \cdot L/m) + 2(q_1 \cdot L/m)^2\right)/2^n
\end{aligned}
$$

Game 6. In this game we modify simulators $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$ to remove their dependency on $\mathcal{F}$ making them self-dependant (they now generate all their responses independently and uniformly at random); the new simulators $S_{2,x}, R_{2,x}, S_{2,x}^{-1}, R_{2,x}^{-1}, S_{2,y}, R_{2,y}, S_{2,y}^{-1}, R_{2,y}^{-1}, S_{2,c}, R_{2,c}, S_{2,c}^{-1}, R_{2,c}^{-1}$ are defined in figures 8 and 9. Unlike $G(3)$ and $G(4)$, where we modified the S and R simulators separately in different games, we had to modify both simulators simultaneously in this game because they are accessing the same shared table $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ and a single simulator is no longer generating a full tuple for every query it receives. That is, if we only modify one of them, the table will suffer from inconsistencies since then one of the simulators will interact with it differently than the other. Although the new simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ still access the tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$, respectively, they no longer check for any failure condition, and they do not need to because they guarantee collision-freeness as we prove below. Also, as stated above, they are not required to generate complete tuples every time they are queried, that is, a query to $S_{2,x}$ or $S_{2,x}^{-1}$ will create the c, y, z fields of a tuple (setting $x = \perp$), a query to $S_{2,y}$ or $S_{2,y}^{-1}$ will create x, c, z (setting $y = \perp$), and a query to $S_{2,c}$ or $S_{2,c}^{-1}$ will create x, y, c (setting $z = \perp$), while a query to $R_{2,x}$ or $R_{2,x}^{-1}$ will create x, c (setting $y = z = \perp$), a query to $R_{2,y}$ or $R_{2,y}^{-1}$ will create y, c (setting $x = z = \perp$), and finally a query to $R_{2,c}$ or $R_{2,c}^{-1}$ will create c, z (setting $x = y = \perp$). Thus, unlike $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}$, with $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ at least two queries are now required in order to create a new complete tuple in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. Since we do not need to check for sequenced and singular queries, as queries from $F_2^{S_2,R_2}$ will now already be in sequence, and direct queries to $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}, i \in \{x, y, c\}$ will automatically be singular (as per their definitions in figures 8 and 9), simulators $S_{2,i}, R_{2,i}^{-1}, S_{2,i}^{-1}, R_{2,i}^{-1}$ will now drop the field t from $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. The easiest way to calculate the distinguishing probability of this game is to observe the differences between the simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}$ and $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$, and demonstrate how changes in the new simulators may affect D's view from $G(5)$ to $G(6)$.

Basically, there are two main differences between the new simulators $S_{2,i}, R_{2,i}, S_{2,i}^{-1}, R_{2,i}^{-1}$ and the old ones $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$:

1. A new single query to the new simulators does not result in creating a full tuple.
2. The c and z values are no longer being generated by a $\mathcal{RO}$, instead they are always being generated uniformly at random.

However, even these changes do not affect D's view in this game because they will not cause collisions. Below, we prove that this game, with the modifications introduced to the simulators, is collision-free.

Lemma 4.9 (Collision freeness of $G(6)$)**.** *Simulators $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$ accessed by $F_{2,i}^{S_{2,i},R_{2,i}}$, where $i \in \{x, y, c\}$, guarantee collision, prefix and fixed-point freeness.*

Simulator $S_{2,x} : (\rightarrow, c, y)$
if $(c_i, y_i, z_i) \in \mathcal{T}_x^K \wedge (c, y) = (c_i, y_i)$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{z}) \cup \mathcal{T}_x^K(\hat{y}) \cup IV\}$

Simulator $S_{2,x}^{-1} : (\leftarrow, c, z)$
if $(c_i, y_i, z_i) \in \mathcal{T}_x^K \wedge (c, z) = (c_i, z_i)$
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{z}) \cup \mathcal{T}_x^K(\hat{y}) \cup IV\}$

Simulator $S_{2,y} : (\rightarrow, x, c)$
if $(x_i, c_i, z_i) \in \mathcal{T}_y^K \wedge (x, c) = (x_i, c_i)$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{z}) \cup \mathcal{T}_y^K(\hat{y}) \cup IV\}$

Simulator $S_{2,y}^{-1} : (\leftarrow, x, z)$
if $(x_i, c_i, z_i) \in \mathcal{T}_y^K \wedge (x, z) = (x_i, z_i)$
  return c_i
else return
  $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{c})\}$

Simulator $S_{2,c} : (\rightarrow, x, y)$
if $(x_i, y_i, c_i) \in \mathcal{T}_c^K \wedge (x_i, y_i) = (x, y)$
  return c_i
else return
  $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{c})\}$

Simulator $S_{2,c}^{-1} : (\leftarrow, x, c)$
if $(x_i, y_i, c_i) \in \mathcal{T}_c^K \wedge (x_i, c_i) = (x, c)$,
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{z}) \cup \mathcal{T}_c^K(\hat{y}) \cup IV\}$

Figure 8: Simulators $S_{2,x}, S_{2,x}^{-1}, S_{2,y}, S_{2,y}^{-1}, S_{2,c}, S_{2,c}^{-1}$

Simulator $R_{2,x:}(\rightarrow, K, x)$
if $(x_i, c_i) \in \mathcal{T}_x^K \wedge x_i = x$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{c})\}$

Simulator $R_{2,y:}(\rightarrow, K, y)$
if $(y_i, c_i) \in \mathcal{T}_y^K \wedge y_i = y$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{c})\}$

Simulator $R_{2,c} : (\rightarrow, K, c)$
if $(c_i, z_i) \in \mathcal{T}_c^K \wedge c_i = c$
  return z_i
else return
  $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{z}) \cup \mathcal{T}_c^K(\hat{y}) \cup IV\}$

Simulator $R_{2,x}^{-1} : (\leftarrow, K, c)$
if $(x_i, c_i) \in \mathcal{T}_x^K \wedge c_i = c$, **return** x_i
else return $x \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_x^K(\hat{x})\}$

Simulator $R_{2,y}^{-1} : (\leftarrow, K, c)$
if $(y_i, c_i) \in \mathcal{T}_y^K \wedge c_i = c$
  return y_i
else return
  $y \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_y^K(\hat{z}) \cup \mathcal{T}_y^K(\hat{y}) \cup IV\}$

Simulator $R_{2,c}^{-1} : (\leftarrow, K, z)$
if $(c_i, z_i) \in \mathcal{T}_c^K \wedge z_i = z$
  return c_i
else return $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_c^K(\hat{c})\}$

Figure 9: Simulators $R_{2,x}, R_{2,x}^{-1}, R_{2,y}, R_{2,y}^{-1}, R_{2,c}, R_{2,c}^{-1}$

Proof. Collisions between two tuples $(x_p, y_p, c_p, z_p), (x_q, y_q, c_q, z_q), p \neq q$, occur if $y_p = z_q$ or $z_p = z_q$ or $y_p = y_q$, while fixed points occur if $y_i = z_i$ or $z_i = IV$, where $i \in \{p, q\}$. We show that as long as $F_{2,i}^{S_{2,i}, R_{2,i}}$ use $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$, where $i \in \{x, y, c\}$, to process any query they receive, the collision and fixed point scenarios above are impossible. The proof follows from the definitions of $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$. Any y_p of any tuple may not collide with any y_q of any other tuple or any z_p of the same tuple, or any z_q of any other tuple because y_p is always begin generated uniformly at random as follows: $y_p \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_i^K(\hat{y}) \cup \mathcal{T}_i^K(\hat{z}) \cup IV\}$, which excludes values of all the y and z fields of the tuples already exist in $\mathcal{T}_i^K, i \in \{x, y, c\}$. Similarly, a new z_i value is generated as follows: $z \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_K(\hat{z}) \cup \mathcal{T}_K(\hat{y}) \cup IV\}$, which again excludes all the

values of the y and z fields of the tuples already exist in $\mathcal{T}_i^K, i \in \{x, y, c\}$ and thus thwarts any possible collision with them. Therefore, collisions between any y and any z in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ are impossible. This also immediately implies that prefix collisions (where $y = z$) are impossible too. Fixed-point-freeness follows since both y and z are generated excluding IV and other existing values of y and z in $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$. □

Consequently, since the view of D will only change if collisions were found in the previous game, the probability in $G(6)$ is the probability that the simulators $S_{1,i}^{\mathcal{F}}, R_{1,i}^{\mathcal{F}}, (S_{1,i}^{\mathcal{F}})^{-1}, (R_{1,i}^{\mathcal{F}})^{-1}, i \in \{x, y, c\}$ in $G(5)$ will output collisions while the modified simulators $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$ in $G(6)$ will not, which is 0. Thus,

$$\begin{aligned} P_6^x = \Pr[D^{F_{2,x}^{S_{2,x},R_{2,x}},S_{2,x},R_{2,x}}] &= P_5^x \\ P_6^y = \Pr[D^{F_{2,y}^{S_{2,y},R_{2,y}},S_{2,y},R_{2,y}}] &= P_5^y \\ P_6^c = \Pr[D^{F_{2,c}^{S_{2,c},R_{2,c}},S_{2,c},R_{2,c}}] &= P_5^c \end{aligned}$$

Game 7. In this game we remove the shared tables $\mathcal{T}_x^K, \mathcal{T}_y^K, \mathcal{T}_c^K$ and modify the simulators $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}, i \in \{x.y, c\}$ to maintain their own separate private tables. The new simulators $S_{3,i}, S_{3,i}^{-1}, R_{3,i}, R_{3,i}^{-1}$ maintain the new tables $\mathcal{T}_{i,S}^K$ (maintained by $S_{2,i}, S_{2,i}^{-1}$), and $\mathcal{T}_{i,R}^K$ (maintained by $R_{2,i}, R_{2,i}^{-1}$), where $i \in \{x, y, c\}$. These new tables contain tuples of the format (c, y, z) for $\mathcal{T}_{x,S}^K, (x, c)$ for $\mathcal{T}_{x,R}^K, (x, c, z)$ for $\mathcal{T}_{y,S}^K, (y, c)$ for $\mathcal{T}_{y,R}^K, (x, y, c)$ for $\mathcal{T}_{c,S}^K, (c, z)$ for $\mathcal{T}_{c,R}^K$. The definitions of the new simulators $S_{3,i}, S_{3,i}^{-1}, R_{3,i}, R_{3,i}^{-1}$ are similar to the definitions of the simulators $S_{2,i}, S_{2,i}^{-1}, R_{2,i}, R_{2,i}^{-1}$ in figures 8 and 9, except that the new simulators now update and refer to their values from their own (unshared) tables. Since the new simulators $S_{3,i}$ have no access to $\mathcal{T}_{i,R}^K$ and $R_{3,i}$ have no access to $\mathcal{T}_{i,S}^K$, one may be inclined to think that $c_p \in \mathcal{T}_{i,S}^K, c_q \in \mathcal{T}_{i,R}^K$ such that $c_p = c_q$ implies a collision. However, this is not the case, the field c here acts as a connecting variable to link the two tables. In fact, $F_2^{S_{3,i},R_{3,i}}$ will always create this linking c among $\mathcal{T}_{i,S}^K$ and $\mathcal{T}_{i,R}^K$ to process its queries.

Lemma 4.10 (Collision freeness within query tables)**.** *Tables $\mathcal{T}_{i,S}^K$ and $\mathcal{T}_{i,R}^K$, maintained separately by simulators $S_{3,i}$ and $R_{3,i}$, respectively, where $i \in \{x, y, c\}$, may not exhibit collisions in the common field c.*

Proof. A genuine collision in the c field means either $\mathcal{T}_{i,S}^K$ has $c_p = c_q$ while $p \neq q$ or $\mathcal{T}_{i,R}^K$ has $c_a = c_b$ while $a \neq b$; but, this cannot happen because both $S_{3,i}$ and $R_{3,i}$ generate c excluding all the other c values of the existing tuples in their respective tables, that is, $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_{i,S}^K(\hat{c})\}$, $c \xleftarrow{\$} \{1,0\}^n \backslash \{\mathcal{T}_{i,R}^K(\hat{c})\}$. Thus, there may at most be one value of c in a table, but that value may exist in both tables which does not imply a collision, rather it acts as a connecting point between the two tables. □

Even though collisions of c may not be encountered, prefix collisions and fixed points are still possible. For example, when $S_{3,y}$ generates z, it has no way to exclude the values of y (to prevent a prefix collision) because there is no y field in $\mathcal{T}_{y,S}^K$ and it has no access to $\mathcal{T}_{y,R}^K$. Similar arguments apply for other simulators (i.e., there is no single table that contains both y and z fields). Also, a fixed point, where $y = z$, cannot be prevented for the same reason. To simulate practical configurations of the ideal cipher and integration function, we also cannot exclude the IV when generating z or y, making a fixed point such as $y = IV$ or $z = IV$ possible. Therefore, the probability in this game is the probability that the simulators will output either

prefix collision, or fixed points. Since the occurrence probability of a prefix collision can be upper bounded by the birthday attack, the overall probability of this game is:

$$\begin{aligned} P_7^x = \Pr[D^{F_{2,x}^{S_{3,x},R_{3,x}},S_{3,x},R_{3,x}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \\ P_7^y = \Pr[D^{F_{2,y}^{S_{3,y},R_{3,y}},S_{3,y},R_{3,y}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \\ P_7^c = \Pr[D^{F_{2,c}^{S_{3,c},R_{3,c}},S_{3,c},R_{3,c}}] &\leq \left((q_1 \cdot L/m + q_2 + q_3)^2 + 2(q_1 \cdot L/m + q_2 + q_3)\right)/2^n \end{aligned}$$

where the 2 instances of $(q_1 \cdot L/m + q_2 + q_3)/2^n$ signify the 2 possible fixed points, namely $y = IV$ or $z = IV$.

Game 8. We can now replace $\mathcal{F}_{2,x}^{S_{3,x},R_{3,x}}, \mathcal{F}_{2,y}^{S_{3,y},R_{3,y}}, \mathcal{F}_{2,c}^{S_{3,c},R_{3,c}}$ by $\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x}, \hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y}, \hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c}$ and $S_{3,x}, S_{3,y}, S_{3,c}$ by $\mathcal{H}_x, \mathcal{H}_y, \mathcal{H}_c$, and $R_{3,x}, R_{3,y}, R_{3,c}$ by $\mathcal{G}_x, \mathcal{G}_y, \mathcal{G}_c$. Clearly, the distinguishing probability of this game is similar to the previous one as the view of D does not change by the replacements as detailed above:

$$\begin{aligned} P_8^x = \Pr[D^{\hat{C}_x^{\mathcal{H}_x,\mathcal{G}_x},\mathcal{H}_x,\mathcal{G}_x} &= 1] = P_7^x \\ P_8^y = \Pr[D^{\hat{C}_y^{\mathcal{H}_y,\mathcal{G}_y},\mathcal{H}_y,\mathcal{G}_y} &= 1] = P_7^y \\ P_8^c = \Pr[D^{\hat{C}_c^{\mathcal{H}_c,\mathcal{G}_c},\mathcal{H}_c,\mathcal{G}_c} &= 1] = P_7^c \end{aligned}$$

Finally, we add the distinguishing probabilities calculated throughout the proof and obtain the full distinguishing bound for each construction. The running time of the simulator is (number of queries) $\times$ (largest query), that is $t_S \leq (q_1 \cdot L/m + q_2 + q_3) \cdot (m + n)$ and is similar in all games. This completes the proof. □

References

[1] Saif Al-Kuwari. *Integrated-key Hash Functions: How to Constructing Keyed Hash Functions from Keyless Ones*, 2011. (manuscript under review). 2, 3

[2] Mihir Bellare, Alexandra Boldyreva, and Adriana Palacio. An Uninstantiable Random-Oracle-Model Scheme for a Hybrid-Encryption Problem. In *Eurocrypt '04*, volume 3027 of *LNCS*, pages 171–188. Springer-Verlag, 2004. 3

[3] Mihir Bellare and Thomas Ristenpart. Hash Functions in the Dedicated-Key Setting: Design Choices and MPP Transforms. In *ICALP '07*, volume 4596 of *LNCS*, pages 399–410. Springer-Verlag, 2007. 2

[4] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: a Paradigm for Designing Efficient Protocols. In *CCS '93*, pages 62–73, 1993. 3

[5] Mihir Bellare and Phillip Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*, 2004. (eprint.iacr.org/2004/331). 6

[6] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. *Journal of the ACM*, 51(4):557–594, 1998. 3

[7] Ran Canetti, Oded Goldreich, and Shai Halevi. On the Random-Oracle Methodology as Applied to Length-Restricted Signature Schemes. In *TCC ' 04*, volume 2951 of *LNCS*, pages 40–57. Springer-Verlag, 2004. 3

[8] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prshant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In *Crypto '05*, volume 3621 of *LNCS*, pages 430–448. Springer-Verlag, 2005. 3, 4, 6

[9] Ivan Damgård. A Design Principle for Hash Functions. In *Crypto '89*, volume 435 of *LNCS*, pages 416–427. Springer-Verlag, 1989. 3

[10] Yevgeniy Dodis, Leo Reyzin, Ronald Rivest, and Emily Shen. Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6. In *FSE '09*, volume 5665 of *LNCS*, pages 104–121. Springer-Verlag, 2009. 2, 3

[11] Daëtan Leurent and Phong Nguyen. How Risky is the Random-Oracle Model? In *Crypto '09*, volume 5677 of *LNCS*, pages 445–464. Springer-Verlag, 2009. 3

[12] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *TCC '04*, volume 2951 of *LNCS*, pages 21–39. Springer-Verlag, 2004. 3

[13] Ralph Merkle. One Way Hash Functions and DES. In *Crypto '89*, volume 435 of *LNCS*, pages 428–446. Springer-Verlag, 1989. 3

[14] Jesper Nielsen. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-Committing Encryption Case. In *Crypto '02*, volume 2442 of *LNCS*, pages 111–126. Springer-Verlag, 2002. 3

[15] Michael Rabin. *Foundations of Secure Computations*, chapter Digitalized Signatures, pages 155–166. Academic Press, 1978. 13

[16] Phillip Rogaway. Formalizing Human Ignorance: Collision-Resistant Hashing Without the Keys. In *Vietcrypt '06*, volume 4341 of *LNCS*, pages 211–228. Springer-Verlag, 2006. 2

[17] X. Wang, D. Feng, X. Lai, and H. Yu. *Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD*, 2004. (eprint.iacr.org/2004/199). 2

[18] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Crypto '05*, volume 3621 of *LNCS*, pages 17–36. Springer-Verlag, 2005. 2

[19] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Eurocrypt '05*, volume 3494 of *LNCS*, pages 19–35. Springer-Verlag, 2005. 2