# Salvaging Indifferentiability in a Multi-stage Setting

Arno Mittelbach

Darmstadt University of Technology, Germany

**Abstract.** The indifferentiability framework by Maurer, Renner and Holenstein (MRH; TCC 2004) formalizes a sufficient condition to safely replace a random oracle by a construction based on a (hopefully) weaker assumption such as an ideal cipher. Indeed, many indifferentiable hash functions have been constructed and could since be used in place of random oracles. Unfortunately, Ristenpart, Shacham, and Shrimpton (RSS; Eurocrypt 2011) discovered that for a large class of security notions, the MRH composition theorem actually does not apply. To bridge the gap they suggested a stronger notion called reset indifferentiability and established a generalized version of the MRH composition theorem. However, as recent works by Demay et al. (Eurocrypt 2013) and Baecher et al. (Asiacrypt 2013) brought to light, reset indifferentiability is not achievable thereby re-opening the quest for a notion that is sufficient for multi-stage games and achievable at the same time.

We present a condition on multi-stage games called *unsplittability*. We show that if a game is unsplittable for a hash construction then the MRH composition theorem can be salvaged. Unsplittability captures a restricted yet broad class of games together with a set of practical hash constructions including HMAC, NMAC and several Merkle-Damgård variants. We show unsplittability for the chosen distribution attack (CDA) game (Bellare et al., Asiacrypt 2009), a multi-stage game capturing the security of deterministic encryption schemes; for message-locked encryption (Bellare et al.; Eurocrypt 2013) a related primitive that allows for secure deduplication; for universal computational extractors (UCE) (Bellare et al., Crypto 2013), a recently introduced standard model assumption to replace random oracles; as well as for the proof-of-storage game given by Ristenpart et al. as a counterexample to the general applicability of the indifferentiability framework.

## 1 Introduction

The notion of indifferentiability, introduced by Maurer, Renner and Holenstein (MRH) [25] can be regarded as a generalization of indistinguishability tailored to situations where internal state is publicly available. It has found wide applicability in the domain of iterative hash functions which are usually built from a fixed-length compression function together with a scheme that describes how arbitrarily long messages are to be processed [26,16,30,23,11]. The MRH

P.Q. Nguyen and E. Oswald (Eds.): EUROCRYPT 2014, LNCS 8441, pp. 603–621, 2014.

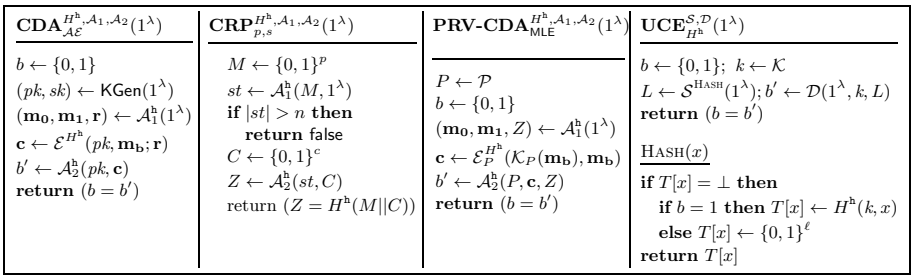


**Fig. 1.** Security Games. From left to right: the chosen distribution attack (CDA) game [4] capturing security in deterministic encryption schemes [3], the proof-of-storage challenge-response game (CRP) due to Ristenpart et al. [29] given as counter-example of the general applicability of the indifferentiability composition theorem, message locked encryption (MLE) [7], and universal computational extractors (UCE) [6] a standard model security assumption on hash-functions.

composition theorem formalizes a sufficient condition under which such a construction can safely instantiate a random oracle: namely indifferentiability of a random oracle. A different view on this is that with indifferentiability one can transfer proofs of security from one idealized setting into a different (and hopefully simpler) idealized setting. For example, proofs in the random oracle model (ROM) [8] imply proofs in the ideal cipher model if a construction from an ideal cipher that is indifferentiable from a random oracle exists.

Ristenpart, Shacham and Shrimpton (RSS) [29] gave the somewhat surprising result that the MRH composition theorem only holds in single-stage settings and does not necessarily extend to multi-stage settings where disjoint adversaries are split over several stages. As counterexample they present a simple *challenge-response game* (CRP, depicted in Figure 1): a file server that is given a file $M$ can be engaged in a simple proof-of-storage protocol where it has to respond with a hash value $\mathcal{H}(M\|C)$ for a random challenge $C$ while only being able to store a short state $st$ (with $|st| \ll |M|$). The protocol can easily be proven secure in the ROM since, without access to file $M$, it is highly improbable for the server to correctly guess the hash value $\mathcal{H}(M||C)$. The server can, however, "cheat" if the random oracle is replaced by one of several indifferentiable constructions. Here the server exploits the internal structure by computing an intermediate chaining value which allows it to later compute extended hash values of the form $H^{\mathsf{h}}(M\|\cdot)$. We refer to [29] for a detailed discussion.

To circumvent the problem of composition in multi-stage settings, RSS propose a stronger form of indifferentiability called *reset indifferentiability* [29], which intuitively states that simulators must be stateless and pseudo-deterministic [2]. While this notion allows composition in any setting, no domain extender can fulfill this stronger form of indifferentiability [17,24,2]. Demay et al. [17] present a second variant of indifferentiability called *resource-restricted indifferentiability* which models simulators with explicit memory restrictions and which lies somewhere in between plain indifferentiability and reset indifferentiability. However, they do not present any positive results such as constructions

that achieve any form of resource-restricted indifferentiability or security games for which a resource-restricted construction allows composition.

The only positive results, we are aware of, is the analysis of RSS of the non-adaptive chosen-distribution attack (CDA) game [4], depicted in Figure 1. CDA captures a security notion for deterministic public-key encryption schemes [3], where the randomness does not have sufficient min-entropy. In the CDA game, the first-stage adversary $\mathcal{A}_1$ outputs two message vectors $\mathbf{m_0}$ and $\mathbf{m_1}$ together with a randomness vector $\mathbf{r}$ which, together, must have sufficient min-entropy independent of the hash functionality. According to a secret bit $b$ one of the two message vectors is encrypted and given, together with the public key, to the second-stage adversary $\mathcal{A}_2$. The adversary wins if it correctly guesses $b$. For the non-adaptive CDA game, RSS give a direct security proof for the subclass of indifferentiable hash functions of the NMAC-type [18], i.e., hash functions of the form $H^{\mathtt{h}}(M) := \mathtt{g}(f^{\mathtt{h}}(M))$ where function $\mathtt{g}$ is a fixed-length random oracle independent of $f^{\mathtt{h}}$ which is assumed to be preimage aware. Note, while this covers some hash functions of interest, it does not, for example, cover chop-MD functions [15] (like SHA-2 for certain parameter settings) or Keccak (aka. SHA-3).

In the lights of the negative results on stronger notions of indifferentiability, we aim at salvaging the current notion; that is, we present tools and techniques to work with plain indifferentiability in multi-stage settings. For this, let us have a closer look at what goes wrong when directly applying the MRH composition theorem in a multi-stage setting.

*Plain Indifferentiability in Multi-stage Settings.* Consider the basic Merkle-Damgård construction[1] and consider a two stage game with adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$. If adversary $\mathcal{A}_1$ makes an $\mathtt{h}$-query $y_1 \leftarrow \mathtt{h}(m_1, \mathcal{IV})$ and passes on this value to adversary $\mathcal{A}_2$, then $\mathcal{A}_2$ can compute arbitrary hash values of the form $m_1 \| \ldots$ without having to know $m_1$. The trick in the MRH composition theorem is to exchange access to $\mathtt{h}$ with access to a simulator $\mathcal{S}$ when placing the adversary in a setting where it plays against the game with random oracle $\mathcal{R}$. If we apply this trick to our two-stage game we need two independent instances of this simulator, one for $\mathcal{A}_1$ and one for $\mathcal{A}_2$. Let's call these $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$. The problem is now, that if $\mathcal{A}_1$ and $\mathcal{A}_2$ do not share sufficient state the same applies to the two simulator instances: they share exactly the same state that is shared between the two adversaries. Thus, if adversary $\mathcal{A}_2$ makes the query $(y, m_2)$ simulator $\mathcal{S}^{(2)}$ does not know that $y$ corresponds to query $(m_1, \mathcal{IV})$ from $\mathcal{A}_1$ and it will thus not be able to answer with a value $y'$ such that $\mathtt{g}(y') = \mathcal{R}(m_1 \| m_2)$. This is, however, expected by $\mathcal{A}_2$ and would be the case if $\mathcal{A}_1$ and $\mathcal{A}_2$ had had access to the deterministic compression function $\mathtt{h}$.

*Contributions.* Our first contribution (Section 3) is to develop a model of hash functions based on directed, acyclic graphs that is rich enough to pinpoint and

[1] The basic MD function $H^{\mathtt{h}}(m_1, \ldots, m_\ell)$ is computed as $H^{\mathtt{h}}(m_1, \ldots, m_\ell) := \mathtt{h}(m_\ell, x_\ell)$ where $x_1 := \mathcal{IV}$ is some initialization vector and $x_{i+1} := \mathtt{h}(m_i, x_i)$.

argue about such problematic adversarial h-queries while at the same time allowing us to consider many different constructions simultaneously. Given this framework we define a property on games and hash functions called UNSPLITTABILITY (Definition 10). If a game is UNSPLITTABLE for a hash construction, this basically means that problematic queries as the one from the above example do not occur.

In Section 4 we then give a composition theorem for UNSPLITTABLE games which intuitively says that if a game is UNSPLITTABLE for an indifferentiable hash construction, then security proofs in the random oracle model carry over if the random oracle is implemented by that particular hash function. Assuming UNSPLITTABILITY, the main technical difficulty in proving composition is to properly derandomize the various simulator instances and make them (nearly) stateless. Note that simulators for indifferentiable hash constructions in the literature are mostly probabilistic and highly stateful. In a multi-stage setting the various instances of the simulator must, however, answer queries consistently, that is, in particular the same query by different adversaries must always be answered with the same answer independent of the order of queries. For this, we build on a derandomization technique developed by Bennet and Gill to show that the complexity classes $\mathcal{BPP}$ and $\mathcal{P}$ are identical relative to a random oracle [10]. One interesting intermediary result is that of a generic indifferentiability simulator that answers queries in a very restricted way.

In Section 5 we show how to prove UNSPLITTABILITY for all multi-stage security games depicted in Figure 1. We show that the CDA game (both, the non-adaptive and adaptive) is UNSPLITTABLE for Merkle-Damgård-like functions as well as for HMAC and NMAC (in the formulation of [5]) thereby complementing the results by RSS. Let us note that, that our results on CDA require less restrictions on the public-key encryption scheme (that is, the encryption scheme does not need to be IND-SIM [29]). Similarly, we show UNSPLITTABILITY for message locked encryption (MLE), a security definition for primitives that allow for secure deduplication [7]. MLE is closely related to CDA with the additional complication that the two adversaries here can communicate "in the clear" via state value $Z$ (see Figure 1). For the RSS proof-of-storage (CRP) game given as counter-example for the general applicability of the MRH composition theorem, we show that it is UNSPLITTABLE for any so-called 2-round hash function. These are hash functions, such as Liskov's Zipper Hash [23] that process the input message twice for computing the final hash value. Finally, we resolve an open problem from [6]. Bellare, Hoang and Keelveedhi (BHK) introduce UCE a standard model assumption for hash constructions which is sufficient to replace a random oracle in a large number of applications [6]. At present the only instantiation of a UCE-secure function is given in the random oracle model and BHK left as open problem whether HMAC can be shown to meet UCE-security assuming an ideal compression function. We show that this is not just the case for HMAC but also for many Merkle-Damgård variants.

Finally, we want to note that we give the results for CDA, MLE and UCE via a meta-result that considers security games for keyed hash functions where

the hash function key is only revealed at the very last stage. We show that all three security games can be subsumed under this class and we show that games from this class are UNSPLITTABLE for a large class of practical hash constructions including HMAC and NMAC and several Merkle-Damgård-like functions such as prefix-free or chop-MD [15]. This is particularly interesting as CDA and MLE are per se not using keyed hash functions, but can be reformulated in this setting and it seems that with keyed hash functions it is simpler to work with indifferentiability in a multi-stage scenario.

## 2 Preliminaries

If $n \in \mathbb{N}$ is a natural number then by $1^n$ we denote the unary representation and by $\langle n \rangle_\ell$ the binary representation of $n$ (using $\ell$ bits). By $[n]$ we denote the set $\{1, 2, \ldots, n\}$. By $\{0,1\}^n$ we denote the set of all bit strings of length $n$ while $\{0,1\}^*$ denotes the set of all finite bit strings. For bit strings $m, m' \in \{0,1\}^*$ we denote by $m||m'$ their concatenation. If $\mathcal{M}$ is a set then by $m \leftarrow \mathcal{M}$ we denote that $m$ was sampled uniformly from $\mathcal{M}$. If $\mathcal{A}$ is an algorithm then by $X \leftarrow \mathcal{A}(m)$ we denote that $X$ was output by algorithm $\mathcal{A}$ on input $m$. As usual $|\mathcal{M}|$ denotes the cardinality of set $\mathcal{M}$ and $|m|$ the length of bit string $m$. Logarithms are to base 2. By $\mathrm{H}_\infty(X)$ we denote the min-entropy of variable $X$, defined as $\mathrm{H}_\infty(X) := \min_x \log(1/\Pr[X = x])$. We assume that any algorithm, game, etc. is implicitly given a security parameter as input, even if not explicitly stated. We call an algorithm *efficient* if its run-time is polynomial in the security parameter. Probability statements of the form $\Pr[\mathrm{step}_1; \mathrm{step}_2 : \mathrm{condition}]$ should be read as the probability that condition holds after the steps are executed in consecutive order. We use standard boolean notation and denote by $\wedge$ the AND by $\vee$ the OR of two values.

*Hash Functions.* A hash function is formally defined as a keyed family of functions $\mathcal{H}(1^\lambda)$ where each key $k$ defines a function $H_k : \{0,1\}^* \to \{0,1\}^n$. "Practical" hash functions are usually built via domain extension from an underlying function $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ that is iterated through an iteration scheme $H$ to process arbitrarily long inputs [26,16,30,23,1,21,31,11,20], with widely varying specifications. The underlying function $\mathtt{h}$ usually is a compression function— the first input taking message blocks and the second an intermediate chaining value—and we will state our results relative to compression functions. As an exception to this rule, the Sponge construction [12] (the design principle behind SHA-3, aka. Keccak [11]) iterates a permutation instead of a compression function. We discuss, how this fits into our model in the full version [27].

*Indifferentiability.* A hash function is called indifferentiable from a random oracle if no distinguisher can decide whether it is talking to the hash function and its ideal compression function or to an actual random oracle and a simulator. We here give the definition of indifferentiability from [15].

**Definition 1.** *A hash construction* $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$*, with black-box access to an ideal function* $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$*, is called indifferentiable from a random oracle* $\mathcal{R}$ *if there exists an efficient simulator* $\mathcal{S}^{\mathcal{R}}$ *such that for any distinguisher* $\mathcal{D}$ *there exists a negligible function* $\mathtt{negl}$*, such that*

$$\left|\Pr\left[\mathcal{D}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda) = 1\right] - \Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}^{\mathcal{R}}}(1^\lambda) = 1\right]\right| \leq \mathtt{negl}(\lambda) \ .$$

*Game Playing.* We use the game-playing technique [9,29] and present here a brief overview of the notation used. A game $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ gets access to adversarial procedures $\mathcal{A}_1,\ldots,\mathcal{A}_m$ and to one or more so called functionalities $\mathcal{F}$ which are collections of two procedures $\mathcal{F}$.hon and $\mathcal{F}$.adv, with suggestive names "honest" and "adversarial". Adversaries (i.e., adversarial procedures) access a functionality $\mathcal{F}$ via the interface exported by $\mathcal{F}$.adv, while all other procedures access the functionality via $\mathcal{F}$.hon. In our case, functionalities are exclusively hash functions which will be instantiated with iterative hash constructions $H^{\mathtt{h}}$. The adversarial interface exports the underlying function $\mathtt{h}$, while the honest interface exports plain access to $H^{\mathtt{h}}$. We thus, instead of writing $\mathcal{F}$.hon and $\mathcal{F}$.adv usually directly refer to $H^{\mathtt{h}}$ and $\mathtt{h}$, respectively. Adversarial procedures can only be called by the game's **main** procedure.

By $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y$ we denote that the game outputs value $y$. If the game is probabilistic or any adversarial procedure is probabilistic then $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ is a random variable and $\Pr\left[G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y\right]$ denotes the probability that the game outputs $y$. By $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)$ we denote that the game is run on random coins $r$.

For this paper we only consider the sub-class of functionality-respecting games as defined in [29]. A game is called *functionality respecting* if only adversarial procedures can call the adversarial interface of functionalities. We define $\mathcal{LG}$ to be the set of all functionality-respecting games. Note that this restriction is a natural restriction if a game is used to specify a security goal in the random oracle model since random oracles do not provide any adversarial interface.

## 3 A Model for Iterative Hash Functions

In the following we present a new model for iterated hash functions that allows to argue about many functions at the same time. A similar endeavor has been made by Bhattacharyya et al. [13] who introduce *generalized domain extension*. For our purpose, we need a more explicit model that allows us to talk about the execution of hash functions in great detail. Still, our model is general enough to capture many different types of constructions, ranging from the plain Merkle-Damgård over variants such as chop-MD [15] to more complex constructions such as NMAC, HMAC [5] or even hash trees. We give an overview over several hash constructions that are captured by our model in the full version of this paper [27].

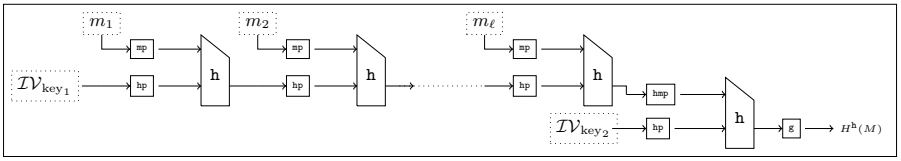


**Fig. 2.** Execution graph for NMAC for message $m_1 \| \ldots \| m_\ell := M$. Value $\mathcal{IV}_{\text{key}_1}$ is an initialization vector representing the first key in the NMAC-construction. Value $\mathcal{IV}_{\text{key}_2}$ is a constant representing the second key. The difference between initialization vectors and constants is that constants are used within the execution graph, i.e., in conjunction with interim values, while initialization vectors are used at the beginning of the graph.

*Execution Graphs - An Introduction.* We model iterative hash functions $H^{\mathtt{h}}$ as directed graphs where each message $M$ is mapped to an execution graph which is constructed independently of a particular choice of function $\mathtt{h}$. Figure 2 presents the *execution graph* for a message $M := m_1 \| \ldots \| m_\ell$ for the NMAC construction [5]. For each input message $M$ the corresponding execution graph represents how the hash value would be computed relative to some oracle $\mathtt{h}$, that is, we require that, relative to an oracle $\mathtt{h}$, a generic algorithm $\mathtt{EVAL}^{\mathtt{h}}$ on input the execution graph for $M$ can then compute value $H^{\mathtt{h}}(M)$. Nodes in the execution graph are either *value-nodes* or *function-nodes*. A value node (indicated by dotted boxes) does not have ingoing edges and the outgoing edge is always labeled with the node's label (possibly prefixed by a constant). Function nodes represent functions and the outgoing edges are labeled with the result of the evaluation of the corresponding function taking the labels of the ingoing edges as input. An $\mathtt{h}$-node represents the evaluation of the underlying function $\mathtt{h}$. Outgoing edges can, thus, only be labeled relative to $\mathtt{h}$. Nodes labeled $\mathtt{mp}, \mathtt{hp}$ or $\mathtt{hmp}$ correspond to preprocessing functions (defined by the hash construction) which ensure that the input to the next $\mathtt{h}$-node is of correct length: $\mathtt{mp}$ processes message blocks, $\mathtt{hp}$ processes $\mathtt{h}$-outputs and $\mathtt{hmp}$, likewise, processes the output of $\mathtt{h}$-nodes but such that it can go into the "message slot" of an $\mathtt{h}$-node (see Figure 2). An execution graph contains exactly one $\mathtt{g}$-node with an unbound outgoing edge which corresponds to an (efficiently) computable transformation such as the identity or truncation.

*Formalizing Hash Functions as Directed Graphs.* We now formalize the above concept to model an iterative hash construction $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ with a compression function of the form $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$. For this let $\mathtt{pad} : \{0,1\}^* \to (\{0,1\}^b)^+$ be a padding function (e.g. Merkle-Damgård strengthening [16,26]) that maps strings to multiples of block size $b$. Let $\mathtt{mp} : \{0,1\}^* \to \{0,1\}^d$, $\mathtt{hp} : \{0,1\}^* \to \{0,1\}^k$ and $\mathtt{hmp} : \{0,1\}^* \to \{0,1\}^d$ be "preprocessing" functions that allow to adapt message blocks and intermediate hash values, respectively. We assume that $\mathtt{pad}, \mathtt{mp}, \mathtt{hp}$, and $\mathtt{hmp}$ are efficiently computable, injective, and efficiently invertible. Note that for many schemes these functions will be the identity function and $b = d$ and $s = k$. Let $\mathtt{g} : \{0,1\}^s \to \{0,1\}^n$

be an efficiently computable transformation (such as the identity function, or a truncation function).[2] Additionally we allow for a dedicated set $\mathcal{IV} \subset \{0,1\}^*$ and containing *initialization vectors* and *constants*.

We give a formal definition of the graph structure in the full version [27] and give here only a quick overview. Execution graphs consist of the following node types: $\mathcal{IV}$-nodes, message-nodes, $\mathtt{h}$-nodes, $\mathtt{mp}$, $\mathtt{hp}$, and $\mathtt{hmp}$-nodes and a single $\mathtt{g}$-node. For each message block $m_1 \| \ldots \| m_\ell := \mathtt{pad}(M)$ the graph contains exactly one message-node. All outgoing edges must again be connected to a node, except for the single outgoing edge of the single $\mathtt{g}$-node. An $\mathtt{h}$-node always has two incoming edges one from an $\mathtt{hp}$-node and one from either an $\mathtt{mp}$ or an $\mathtt{hmp}$-node. Message nodes can be connected to $\mathtt{mp}$-nodes. The outbound edges from $\mathtt{h}$ can be connected to either $\mathtt{hp}$ or $\mathtt{hmp}$-nodes.[3] A *valid execution graph* is a non-empty graph that complies with the above rules. We require that for each message $M \in \{0,1\}^*$ there is exactly one valid execution graph and that there is an efficient algorithm that given $M$ constructs the execution graph.

Besides valid execution graphs we introduce the concept of *partial execution graphs* which are non-empty graphs that comply to the above rules with the only exception that they do not contain a $\mathtt{g}$-node. Hence, they contain exactly one unbound outgoing edge from an $\mathtt{h}$-node. A partial execution graph is always a sub-graph of potentially many valid execution graphs. Given a valid execution graph a partial execution graph can be constructed by choosing an $\mathtt{h}$-node and removing every node that can be reached via directed path from that $\mathtt{h}$-node and then remove all unconnected components that do not have a directed path to the chosen $\mathtt{h}$-node.

We define $\mathtt{EVAL}$ to be a generic, deterministic algorithm evaluating execution graphs relative to an oracle $\mathtt{h}$. Let $\mathfrak{eg}$ be a valid execution graph for some message $M \in \{0,1\}^*$. To evaluate $\mathfrak{eg}$ relative to oracle $\mathtt{h}$, algorithm $\mathtt{EVAL}^{\mathtt{h}}(\mathfrak{eg})$ recursively performs the following steps: search for a node that has no inbound edges or for which all inbound edges are labeled. If the node is a function-node then evaluate the corresponding function using the labels from the inbound edges as input. If the node is a value-node, use the corresponding label as result. Remove the node from the graph and label all outgoing edges with the result. If the last node in the graph was removed stop and return the result. Note that $\mathtt{EVAL}^{\mathtt{h}}(\mathfrak{eg})$ runs in time at most $\mathcal{O}\left(|V^2|\right)$ assuming that $\mathfrak{eg}$ contains $|V|$ many nodes. If $\mathfrak{pg}$ is a partial execution graph then $\mathtt{EVAL}^{\mathtt{h}}(\mathfrak{pg})$, likewise, computes the partial graph outputting the result of the final $\mathtt{h}$-node. We denote by $\mathtt{g}(\mathfrak{pg})$ the corresponding execution graph where the single outbound $\mathtt{h}$-edge of $\mathfrak{pg}$ is connected to a $\mathtt{g}$-node. We call this the *completed* execution graph for $\mathfrak{pg}$.

We can now go on to define iterative hash functions such as Merkle-Damgård-like functions. Informally, an iterative hash function consists of the definitions

[2] We stress that $\mathtt{g}$ is efficiently computable and not an independent (ideal) compression function.

[3] The difference between $\mathtt{hp}$ and $\mathtt{hmp}$ is that $\mathtt{hp}$ outputs values in $\{0,1\}^k$ which $\mathtt{hmp}$ outputs values in $\{0,1\}^d$. Note that function $\mathtt{h}$ is defined as $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$.

of the preprocessing functions, the padding function and the final transformation $\mathtt{g}(\cdot)$. Furthermore, we require (efficient) algorithms that construct execution graphs as well as parse an execution graph to recover the corresponding message.

**Definition 2.** *Let $\mathcal{IV} \subset \{0,1\}^*$ be a set of named initialization vectors and $|\mathcal{IV}|$ be polynomial in the security parameter $\lambda$. We say $H^{\mathtt{h}}_{\mathtt{g},\mathtt{mp},\mathtt{hp},\mathtt{hmp},\mathtt{pad}} : \{0,1\}^* \to \{0,1\}^n$ is an iterative hash function if there exist deterministic and efficient algorithms* `construct` *and* `extract` *as follows:*

`construct`**:** *On input $M \in \{0,1\}^*$, algorithm* `construct` *outputs a valid execution graph containing one message-node for every block in $m_1 \| \ldots \| m_\ell := \mathtt{pad}(M)$. For all messages $M \in \{0,1\}^*$ it holds that $H^{\mathtt{h}}(M) = \mathtt{EVAL}^{\mathtt{h}}(\mathtt{construct}(M))$. For any two $M, M' \in \{0,1\}^*$ with $|M| = |M'|$ it holds that graphs $\mathtt{construct}(M)$ and $\mathtt{construct}(M')$ are identical but for labels of message-nodes.*[4]

`extract`**:** *On input a valid execution graph $\mathfrak{cg}$, algorithm* `extract` *outputs message $M \in \{0,1\}^*$ if, and only if, $\mathtt{construct}(M)$ is identical to $\mathfrak{cg}$. On input a partial execution graph $\mathfrak{pg}$, algorithm* `extract` *outputs message $M \in \{0,1\}^*$ if, and only if, the completed execution graph $\mathtt{g}(\mathfrak{pg})$ is identical to $\mathtt{construct}(M)$. Otherwise* `extract` *outputs $\perp$.*

*When functions $\mathtt{g}, \mathtt{mp}, \mathtt{hp}, \mathtt{hmp}$ and* `pad` *are clear from context we simply write $H^{\mathtt{h}}$.*

We give a detailed description of valid execution graphs, extensions to the model that, for example, cover keyed constructions, as well as several examples of hash constructions that are covered by Definition 2 in the full version [27].

### 3.1 Important h-Queries

Considering the execution of hash functions as graphs allows us to identify certain types of "important" queries by their position in the graph relative to a function $\mathtt{h}$. Assume that $Q = (m_i, x_i)_{1 \le i \le p}$ is an ordered sequence of $\mathtt{h}$-queries to compression function $\mathtt{h}$. If we consider the $i$-th query $q_i = (m_i, x_i)$ then only queries appearing before $q_i$ in $Q$ are relevant for our upcoming naming conventions. We call $q_i$ an *initial query* if, and only if, $\mathtt{hp}^{-1}(x_i) \in \mathcal{IV}$. Besides initial queries we are interested in queries that occur "in the execution graph" and we call these *chained queries*. We call query $q_i$ a *chained query* if given the queries appearing before $q_i$ there exists a valid (partial) execution graph containing an $h$-node with its unbound edge labeled with value $\mathtt{hp}^{-1}(x_i)$. Finally, we call query $q_i$ result query for message $M$, if $\mathtt{g}(q_i) = H^{\mathtt{h}}(M)$ and $q_i$ is a chained query. We define result queries in a broader sense and independent of a specific message by considering all possible partial graphs induced by query set $Q$ and say that a query is a result query if it is a chained query and if its induced partial graph $\mathfrak{pg}$ can be completed to a valid execution graph, that is, $\mathtt{g}(\mathfrak{pg})$ is a valid execution graph. For a visualization of the query types see Figure 3.

[4] This condition ensures that the graph structure does not depend on the content of messages but only on its length.

**Definition 3.** *Let $Q = (m_i, x_i)_{1 \le i \le p}$ be a sequence of queries to $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$. Let $q_i = (m_i, x_i)$ be the $i$-th query in $Q$ and let $Q_{|_{1,\ldots,i}}$ denote the sequence $Q$ up to and including the $i$-th query. Let the predicate $\mathsf{init}(q_i) := \mathsf{init}(m_i, x_i)$ be true if, and only if, $\mathtt{hp}^{-1}(x_i) \in \mathcal{IV}$. We define the predicate $\mathsf{chained}^Q(m_i, x_i)$ to be true if, and only if,*

$$\mathsf{init}(m_i, x_i) \quad \vee \quad \exists\, j \in [i-1] : \Big(\mathsf{chained}^Q(m_j, x_j) \wedge \mathtt{hp}(\mathtt{h}(m_j, x_j)) = x_i\Big) \ .$$

*Let $\mathfrak{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i]$ denote the set of partial graphs such that for all $\mathfrak{pg} \in \mathfrak{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i]$ it holds that all $\mathtt{h}$ queries occurring during the computation of $\mathtt{EVAL}^{\mathtt{h}}(\mathfrak{pg})$ are in $Q_{|_{1,\ldots,i}}$ and that the final $\mathtt{h}$-query equals $q_i$.[5] We define the predicate $\mathsf{result}^Q(m_i, x_i)$ to be true if, and only if,*

$$\mathsf{chained}^Q(m_i, x_i) \quad \wedge \quad \exists \mathfrak{pg} \in \mathfrak{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i] : \mathtt{g}(\mathfrak{pg}) \text{ is a valid execution graph} \ .$$

*We drop the reference to the query set $Q$ if it is clear from context.*

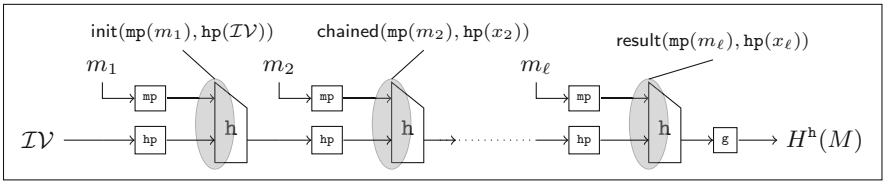


**Fig. 3.** Denoting queries in the Merkle-Damgård construction where value $x_2$ is computed as $x_2 := \mathtt{h}(\mathtt{mp}(m_1), \mathtt{hp}(\mathcal{IV}))$ and value $x_l$ is computed recursively as $x_\ell := \mathtt{h}(\mathtt{mp}(m_\ell), \mathtt{hp}(x_{\ell-1}))$

### 3.2 Message Extractors and Missing Links

We now give two important lemmas concerning iterative hash functions. The first argues that if an adversary does not make all $\mathtt{h}$-queries in the computation of $H^{\mathtt{h}}(M)$ for some message $M$, then its probability of computing the corresponding hash value is small. To get an intuition note that each $\mathtt{h}$-node has a directed path to the final $\mathtt{g}$-node. As we model the underlying function as ideal, an $\mathtt{h}$-evaluation has $s$ bits of min-entropy which are, so to speak, sent down the network to the final $\mathtt{g}$-node. We refer to the full version [27] for the proof.

**Lemma 1.** *Let function $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ be an iterative hash function and let $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ be a fixed-length random oracle. Let $\mathcal{A}^{\mathtt{h}}$ be an adversary that makes at most $q_{\mathcal{A}}$ many queries to $\mathtt{h}$. Let $\mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^\lambda; r))$ denote the adversary's queries to oracle $\mathtt{h}$ when algorithm $\mathcal{A}$ runs on randomness $r$ and*

[5] If $\mathtt{h}$ is modeled as an ideal function then set $\mathfrak{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i]$ contains with very high probability at most one partial graph as multiple graphs induce collisions on $\mathtt{h}$.

*by* $\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))$ *denote the* $\mathtt{h}$*-queries during the evaluation of* $H^{\mathtt{h}}(M)$*. Then it holds that*

$$\Pr_{r,\mathtt{h}}\left[(M,y) \leftarrow \mathcal{A}^{\mathtt{h}}(1^{\lambda};r) : \begin{array}{r} H^{\mathtt{h}}(M) = y \quad \wedge \\ \left(\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M)) \setminus \mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^{\lambda};r))\right) \neq \emptyset \end{array}\right] \leq \frac{q_{\mathcal{A}}}{2^s} + \frac{1}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))}}$$

*where* $\setminus$ *denotes the simple complement of sets and* $U_s$ *denotes a random variable uniformly distributed in* $\{0,1\}^s$*. The probability is over the choice of random oracle* $\mathtt{h}$ *and the coins of* $\mathcal{A}$*.*

Next, we show that given the sequence of $\mathtt{h}$-queries and corresponding answers of an adversary, there exists an efficient and deterministic extractor $\mathcal{E}$ that can reconstruct precisely the set of messages for which the adversary "knows" the corresponding hash value. We refer to the full version [27] for the proof.

**Lemma 2.** *Let function* $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ *be an iterative hash function and* $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ *a fixed-length random oracle. Let* $\mathcal{A}^{\mathtt{h}}$ *be an adversary making at most* $q_{\mathcal{A}}$ *queries to* $\mathtt{h}$*. Let* $\mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^{\lambda};r))$ *denote the adversary's queries to oracle* $\mathtt{h}$ *(together with the corresponding oracle answer) when algorithm* $\mathcal{A}$ *runs on randomness* $r$*. Then there exists an efficient deterministic extractor* $\mathcal{E}$ *outputting sets* $\mathcal{M}$ *and* $\mathcal{Y}$ *with* $|\mathcal{M}| = |\mathcal{Y}| \leq 3q_{\mathcal{A}}$*, such that*

$$\Pr_{r,\mathtt{h}}\left[\begin{array}{r} (M,y) \leftarrow \mathcal{A}^{\mathtt{h}}(1^{\lambda};r); \\ (\mathcal{M},\mathcal{Y}) \leftarrow \mathcal{E}(\mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^{\lambda};r))) \end{array} : \begin{array}{l} \exists\ X \in \mathcal{M} : H^{\mathtt{h}}(X) \notin \mathcal{Y} \quad \vee \\ \left(H^{\mathtt{h}}(M) = y \wedge M \notin \mathcal{M}\right) \end{array}\right] \leq \frac{3q_{\mathcal{A}}^2}{2^{\mathrm{H}_\infty(g(U_s))}}\ .$$

*Value* $U_s$ *denotes a random variable uniformly distributed in* $\{0,1\}^s$*. The probability is over the coins* $r$ *of* $\mathcal{A}^{\mathtt{h}}$ *and the choice of random oracle* $\mathtt{h}$*.*

### 3.3 $\mathtt{h}$-Queries during Functionality Respecting Games

We now define various terms that allow us to talk about specific queries from adversarial procedures to the underlying function $\mathtt{h}$ of iterative hash function $H^{\mathtt{h}}$ during game $G$. Recall that, as do Ristenpart et al. [29], we only consider the class of *functionality-respecting* games (see Section 2) where only adversarial procedures may call the adversarial interface of functionalities (i.e., the underlying function $\mathtt{h}$ in our case).

**Definition 4.** *Let* $G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ *be a functionality respecting game with access to hash functionality* $H^{\mathtt{h}}$ *and adversarial procedures* $\mathcal{A}_1,\ldots,\mathcal{A}_m$*. We denote by* $\mathsf{qry}^{G,\mathtt{h}}$ *the sequence of queries to the adversarial interface of* $H^{\mathtt{h}}$ *(that is,* $\mathtt{h}$*) during the execution of game* $G$*.*

Note that $\mathsf{qry}^{G,\mathtt{h}}$ is a random variable over the random coins of game $G$. Thus, we can regard the query sequence as a deterministic function of the random coins. In this light, in the following we define subsequences of queries belonging to certain adversarial procedures such as the $i$-th query of the $j$-th adversarial procedure.

Game $G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ can call adversarial procedures $\mathcal{A}_1,\ldots,\mathcal{A}_m$ in any order and multiple times. Thus, we first define a mapping from the sequence of

adversarial procedure calls by the game's main procedure to the actual adversarial procedure $\mathcal{A}_i$. For better readability, we drop the superscript identifying game G in the following definitions and whenever the game is clear from context. We drop the superscript identifying oracle $\mathtt{h}$ exposed by the adversarial interface of functionality $H^{\mathtt{h}}$ if clear from context.

**Definition 5.** *We define* $\mathsf{AdvSeq}_i$ *(for* $i \geq 1$*) to denote the adversarial procedure corresponding to the i-th adversarial procedure call by game G. We set* $|\mathsf{AdvSeq}|$ *to denote the total number of adversarial procedure calls by G.*

We define sequence of $\mathtt{h}$-queries made by the $i$-th adversarial procedure $\mathsf{AdvSeq}_i$ as:

**Definition 6.** *By* $\mathsf{qry}_i$ *we denote the sequence of queries to* $\mathtt{h}$ *by procedure* $\mathsf{AdvSeq}_i$ *during the i-th adversarial procedure call by the game's main procedure. By* $\mathsf{qry}_{i,j}$ *we denote the j-th query in this sequence.*

We also need a notion which captures all those queries executed before a specific adversarial procedure $\mathsf{AdvSeq}_i$ was called. For this, we will slightly abuse notation and "concatenate" two (or more) sequences, i.e., if $S_1$ and $S_2$ are two sequences, then by $S_1||S_2$ we denote the sequence that contains all elements of $S_1$ followed by all elements of $S_2$ in their specific order.

**Definition 7.** *By* $\mathsf{qry}_{<i}$ *we denote the sequence of queries to* $\mathtt{h}$ *before the execution of procedure* $\mathsf{AdvSeq}_i$*. By* $\mathsf{qry}_{<i,j}$ *we denote the sequence of queries to* $\mathtt{h}$ *up to the j-th query of the i-th adversarial procedure call. Formally,*

$$\mathsf{qry}_{<i} := \Big|\Big|_{k=1}^{i-1} \mathsf{qry}_k \qquad \textit{and} \qquad \mathsf{qry}_{<i,j} := \mathsf{qry}_{<i} \,||\, \Big|\Big|_{k=1}^{j-1} \mathsf{qry}_{i,k}$$

Finally, we define the sequence of $\mathtt{h}$-queries by procedure $\mathsf{AdvSeq}_i$ up-to the $i$-th adversarial procedure call by the game's main procedure. That is, in addition to queries $\mathsf{qry}_i$ we have all queries from previous calls to $\mathsf{AdvSeq}_i$ by the game's main procedure.

**Definition 8.** *By* $\mathsf{qry}_{<\mathcal{A}_i,j}$ *we denote the sequence of queries to procedure* $\mathtt{h}$ *by the i-th adversarial procedure* $\mathsf{AdvSeq}_i$ *up-to query* $\mathsf{qry}_{<i,j}$*. Formally,*

$$\mathsf{qry}_{<\mathcal{A}_i,j} := \Big|\Big|_{\substack{0<\ell<i,\\ \mathsf{AdvSeq}_\ell=\mathsf{AdvSeq}_i}} \mathsf{qry}_\ell \;\;||\;\; \Big|\Big|_{k=1}^{j-1} \mathsf{qry}_{i,k} \;.$$

*Bad Result Queries.* Having defined queries to the adversarial interface of the hash functionality (i.e., underlying function $\mathtt{h}$) occurring during a game $G$ allows us to use our notation established in Section 3.1 on $\mathtt{h}$-queries: initial queries, chained queries and result queries. For example, we can say that query $\mathsf{qry}_{i,j}$ is an initial query. With this, we now define a $\mathsf{bad}$ event corresponding to splitting

up the evaluation of hash values via several adversarial stages (also refer to the introduction).

Informally, we call a query $(m, x)$ to function $\mathtt{h}(\cdot,\cdot)$ $\mathsf{badResult}$ if it is a result query (cp. Definition 3) with respect to **all** previous queries during the game, but it is not a chained query (and thus not a result query) if we restrict the sequence of queries to that of the current adversarial procedure. Note that, whether or not a query is $\mathsf{bad}$ only depends on queries to $\mathtt{h}$ prior to the query in question and is not changed by any query coming later in the game. (Note the change in the underlying sequence for the two predicates in the following definition.)

**Definition 9.** *Let* $G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ *be any game. Let* $(m, x) := \mathsf{qry}_{i,j}$ *be the* $j$*-th query to function* $\mathtt{h}$ *by adversary* $\mathsf{AdvSeq}_i$*. Then query* $(m, x)$ *is called* $\mathsf{badResult}^{\mathcal{A}_i}(\mathsf{qry}_{i,j})$ *if, and only if:* $\mathsf{result}^{\mathsf{qry}_{<i,j}}(m, x)$ *and* $\neg\mathsf{chained}^{\mathsf{qry}_{<\mathcal{A}_i,j}}(m, x)$.

## 4 Unsplittable Multi-stage Games

The formalization of hash functions together with terminology on particular queries during a game allows us to define a property on games that will be sufficient to argue composition similar to that of the MRH composition theorem for indifferentiability. We call a game $G \in \mathcal{LG}$ UNSPLITTABLE for an iterative hash construction $H^{\mathtt{h}}$, if two conditions hold: 1) For any adversary $\mathcal{A}_1, \ldots, \mathcal{A}_m$ there exists adversary $\mathcal{A}_1^*, \ldots, \mathcal{A}_m^*$ such that games $G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ and $G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}$ change only by a small factor, and 2) During game $G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}$ we have that bad result queries only occur with small probability. Intuitively, this means that it does not help adversaries to split up the computation of hash values over several distinct adversarial procedures. After formally defining unsplittability we will then formulate the accompanying composition theorem which informally states that if a game is UNSPLITTABLE for an indifferentiable hash construction $H^{\mathtt{h}}$, then security proofs in the ROM carry over if the random oracle is implemented by that particular hash function.

**Definition 10.** *Let* $H^{\mathtt{h}}$ *be an iterative hash function and let* $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ *be an ideal function. We say a functionality respecting game* $G \in \mathcal{LG}$ *is* $(t_{\mathcal{A}^*}, q_{\mathcal{A}^*}, \epsilon_G, \epsilon_{\mathsf{bad}})$-UNSPLITTABLE *for* $H^{\mathtt{h}}$ *if for every adversary* $\mathcal{A}_1, \ldots, \mathcal{A}_m$ *there exists algorithm* $\mathcal{A}_1^*, \ldots, \mathcal{A}_m^*$ *such that for all values* $y$

$$\Pr\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y\right] \leq \Pr\left[G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*} \Rightarrow y\right] + \epsilon_G \ .$$

*Adversary* $\mathcal{A}_i^*$ *has run-time at most* $t^*_{\mathcal{A}_i}$ *and makes at most* $q^*_{\mathcal{A}_i}$ *queries to* $\mathtt{h}$*. Moreover, it holds for game* $G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}$ *that:*

$$\Pr\left[\exists i \in [|\mathsf{AdvSeq}|], \exists j \in [q^*_{\mathcal{A}_i}] \ : \mathsf{badResult}^{\mathcal{A}_i}(\mathsf{qry}_{i,j})\right] \leq \epsilon_{\mathsf{bad}} \ .$$

*The probability is over the coins of game* $G^{H^{\mathtt{h}},\mathcal{A}^*{}_1,\ldots,\mathcal{A}^*{}_m}$ *and the choice of function* $\mathtt{h}$*.*

### 4.1 Composition for Unsplittable Multi-stage Games

We here give the composition theorem for UNSPLITTABLE games in the asymptotic setting. The full theorem with concrete advantages is given in the full version [27]. Due to space limitations, we here also only present a much shortened proof sketch.

**Theorem 1 (Asymptotic Setting).** *Let $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ be an iterative hash function indifferentiable from a random oracle $\mathcal{R}$ and let $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ be an ideal function. Let game $G \in \mathcal{LG}$ be any functionality respecting game that is* UNSPLITTABLE *for $H^{\mathtt{h}}$ and let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be an adversary. Then, there exists efficient adversary $\mathcal{B}_1, \ldots, \mathcal{B}_m$ and negligible function* $\mathtt{negl}$ *such that for all values $y$*

$$\left|\Pr\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y\right] - \Pr\left[G^{\mathcal{R},\mathcal{B}_1,\ldots,\mathcal{B}_m} \Rightarrow y\right]\right| \leq \mathtt{negl}(\lambda) \ .$$

*Proof (Proof Sketch).* The proof consists of two steps. In a first step we are going to take the indifferentiability simulator for $H^{\mathtt{h}}$ and transform it into a simulator with a special structure that we call $\mathcal{S}_d$. Secondly, we take the UNSPLITTABILITY-property of game $G$ to get a set of adversaries $\mathcal{A}_1^*, \ldots, \mathcal{A}_m^*$ such that during game $G^{\mathcal{F},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}$ bad result queries (cp. Definition 9) occur only with negligible probability. This property, together, with the structure of simulator $\mathcal{S}_d$ then allows to argue composition, similarly to RSS in their composition theorem for reset-indifferentiability: Theorem 6.1 in [28]. (Theorem 4 in the proceedings version [29]).

*Construction of $\mathcal{S}_d$.* We begin with the construction of simulator $\mathcal{S}_d$. Since $H^{\mathtt{h}}$ is indifferentiable from a random oracle there exists a simulator $\mathcal{S}$ such that no efficient distinguisher $\mathcal{D}$ can distinguish between talking to $(H^{\mathtt{h}}, \mathtt{h})$ or $(\mathcal{R}, \mathcal{S}^{\mathcal{R}})$. From this simulator we are going to construct a generic simulator $\mathcal{S}_*$ which keeps track of all queries internally constructing any potential partial graph for the query-sequence. We give a shortened description of simulator $\mathcal{S}_*$ in Figure 4. If a query corresponds to a result query (cp. Definition 3) it ensures to be compatible with the random oracle by picking a value from the preimage of $\mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg})))$ uniformly at random (see line 8), where $\mathtt{pg}$ is the corresponding partial graph. Note that this ensures consistency with the answers of the random oracle. Otherwise, if the query is not a result query, it simply responds with a random value (line 9). The full construction and proof of indifferentiability is presented in the full version [27].

In a next step (the details are given in [27]) we derandomize simulator $\mathcal{S}_*$ using the random oracle and a derandomization technique by Bennet and Gill [10]. For any fixed value $t_{\mathcal{D}}$, this yields simulator $\mathcal{S}_d$ such that for any distinguisher $\mathcal{D}$ that runs in time at most $t_{\mathcal{D}}$ it holds that

$$\left|\Pr\left[\mathcal{D}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda) = 1\right] - \Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}_d^{\mathcal{R}}}(1^\lambda) = 1\right]\right| \leq \mathtt{negl}(\lambda) \ .$$

**Simulator** $\mathcal{S}_*(m,x)$ :
1 **if** $\mathcal{M}[m,x] \neq \perp$ **then return** $\mathcal{M}[m,x]$
2 $\mathcal{T} \leftarrow \{\}$
3 **if** $\mathsf{init}(m,x)$ **then**
4 create partial graph from $(m,x)$ and add to $\mathcal{T}$
5 test all existing partial graphs, if any can be extended
6 by query$(m,x)$. If so, add result to $\mathcal{T}$
7 **if** $\exists \mathtt{pg} \in \mathcal{T} : \mathtt{extract}(\mathtt{pg}) \neq \perp$ **then**
8 $\mathcal{M}[m,x] \leftarrow\$\ \mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg})))$
9 **else** $\mathcal{M}[m,x] \leftarrow\$\ \{0,1\}^s$
10 **if** $|\mathcal{T}| > 0$ **then**
11 label output edge of any graph in $\mathcal{T}$ by $\mathcal{M}[m,x]$
12 add all graphs in $\mathcal{T}$ to a list of partial graphs
13 **return** $\mathcal{M}[m,x]$

**Fig. 4.** Simulator $\mathcal{S}_*$ for proof of Theorem 1. $\mathcal{S}_*$ maintains a list of partial graphs that can be constructed from the query sequence. If query $(m,x)$ is an initial query it constructs the corresponding partial graph and adds it to the temporary set $\mathcal{T}$. It then tries all existing partial graphs, if they can be extended by the current query. A query is answered either by a random value or (for result queries) by sampling a value uniformly at random from $\mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg}))$.

*Using $\mathcal{S}_d$ with* UNSPLITTABLE *Games.* Let $\mathcal{A}_1^*, \ldots, \mathcal{A}_m^*$ be such that during game $G^{\mathcal{F},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}$ bad result queries occur only with negligible probability. We now set $\mathcal{B}_i := \mathcal{A}_i^{*\mathcal{S}_d^{(i)}}$ where every $\mathcal{S}_d^{(i)}$ denotes an independent copy of $\mathcal{S}_d$. The structure of $\mathcal{S}_d$ ensures that non-result queries (cp. Definition 3) are answered consistently over the several independent copies. Furthermore, the fact that result queries are with overwhelming probability not bad ensures that also these are answered consistently. We, thus, get that

$$\Pr\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y\right] \approx \Pr\left[G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*} \Rightarrow y\right] \approx \Pr\left[G^{\mathcal{R},\mathcal{A}_1^{*\mathcal{S}_d^{(1)\mathcal{R}}},\ldots,\mathcal{A}_m^{*\mathcal{S}_d^{(m)\mathcal{R}}}} \Rightarrow y\right]$$

which yields that $\Pr\left[G^{\mathcal{R},\mathcal{B}_1,\ldots,\mathcal{B}_m} \Rightarrow y\right] \leq \mathtt{negl}$. □

## 5 Applications

We turn to the task of proving UNSPLITTABILITY for the various multi-stage games from the introduction: While for the RSS proof-of-storage game we will give a direct proof (which appears only in the full version [27]) we prove the results for CDA, MLE and UCE via a meta result on games using keyed hash functions (Theorem 2).

### 5.1 Unsplittability of Keyed-Hash Games

Let $\mathsf{qry}^{H^{\mathtt{h}}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)\right]$ be the list of queries by game $G$ (running on random coins $r$) to the honest interface of the functionality (i.e., $H^{\mathtt{h}}$) and let

$$\mathsf{qry}^{\mathtt{h}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)\right] := \left\{(m,x) : \exists M \in \mathsf{qry}^{H^{\mathtt{h}}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)\right],\ (m,x) \in \mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))\right\}$$

be the list of queries by game $G$, when run on random coins $r$, to $\mathtt{h}$ triggered by queries to the honest interface of the functionality. (Note that the adversarial procedures $\mathcal{A}_1, \ldots, \mathcal{A}_m$ never query the honest interface.) For fixed random coins $r$ and an adversarial $\mathtt{h}$-query $\mathsf{qry}_{i,j}$ during game $G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)$ we set

$$G\text{-}\mathsf{relevant}(\mathsf{qry}_{i,j}; r) \iff \mathsf{qry}_{i,j} \in \mathsf{qry}^{\mathtt{h}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)\right]$$

That is, we call an adversarial query $G$-relevant if the same query occurs during the honest computation of an $H^{\mathtt{h}}$ query by game G.

Let us observe that we can replace the adversarial interface $\mathtt{h}$ given to an adversarial procedure by one that differs from $\mathtt{h}$ on all points except for points that are also queried indirectly by the game (i.e., queries which are $G$-relevant), without changing the outcome of the game (or rather its distribution over the choice of ideal functionality $\mathtt{h}$).

*Keyed-Hash Games.* Hash functions can be considered in a keyed setting, where a key is included in the computation of every hash value. HMAC or NMAC were designed as keyed functions, other hash functions like Merkle-Damgård variants can be adapted to the keyed setting, for example, by requiring that the key is prepended to the message. In the following we write $H^{\mathtt{h}}(\kappa, M)$ to denote an iterative hash construction with an explicit key input (for further information on how keyed hash constructions are captured by our framework we refer to the full version [27]).

Many keyed constructions are designed such that the key is used in all initial queries. HMAC and NMAC are of that type, and also the adapted Merkle-Damgård variants such as chop-MD or prefix-free-MD [15] can be regarded of that type, if the key is always prepended to the message. We call such hash functions *key-prefixed hash functions.*

**Definition 11.** *A keyed iterative hash function $H^{\mathtt{h}}$ is called* key-prefixed*, if for all $\kappa \in \mathcal{K}$ and all $M \in \{0,1\}^*$*

$$\forall (m, x) \in \mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(\kappa, M)) : \neg\mathsf{init}(m, x) \vee \mathtt{mp}^{-1}(m) = \kappa \vee \mathtt{hp}^{-1}(x) = \kappa$$

*where $\mathcal{K}$ denotes the key-space of function $H^{\mathtt{h}}$.*

Now, consider games that only make keyed hash queries. By this we mean that either the game is defined using keyed hash functions directly, or it can be restated as such by identifying a part of each query as key, for example, because some parameter is prepended to every hash query.

**Definition 12.** *We call a game $G \in \mathcal{LG}$ a* keyed-hash game*, if $G$ only makes keyed hash queries. We denote by $\mathcal{K}_G[H^{\mathtt{h}}, r]$ the set of keys used by $G$ when run on coins $r$ and with hash function $H^{\mathtt{h}}$, and require that $\mathcal{K}_G[H^{\mathtt{h}}, r]$ is polynomially bounded and chosen independently of the adversarial procedures.*

We now show that an interesting sub-class of *keyed-hash games* are UNSPLITTABLE for *key-prefixed hash functions.*

**Theorem 2.** *Let $G \in \mathcal{LG}$ be a keyed-hash game where adversarial procedures $\mathcal{A}_1, \ldots, \mathcal{A}_m$ are called exactly once and in this order. Let $H^{\mathtt{h}}$ be a key-prefixed iterative hash-function, that is indifferentiable from a random oracle. Let $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ be an ideal function. Denote by $\mathsf{View}[\mathcal{A}_i; H^{\mathtt{h}}, r]$ the view of adversary $\mathcal{A}_i$, i.e., the random coins of $\mathcal{A}_i$ together with its input and answers to any of its oracle queries when game $G$ is run with coins $r$ and function $H^{\mathtt{h}}$.*

*If for every efficient extractor $\mathcal{E}$ and for every efficient adversary $\mathcal{A}_i$ (for $i = 1, \ldots, m-1$) there exists negligible function $\mathtt{negl}$ such that*

$$\Pr_r \left[ k \leftarrow \mathcal{E}(\mathsf{View}[\mathcal{A}_i; H^{\mathtt{h}}, r]) : k \in \mathcal{K}_G[H^{\mathtt{h}}, r] \right] \leq \mathtt{negl}(\lambda)$$

*and adversary $\mathcal{A}_m$ gets $\mathcal{K}_G[H^{\mathtt{h}}, r]$ as part of its input then $G$ is* UNSPLITTABLE *for $H^{\mathtt{h}}$.*

The theorem can be applied to the CDA, the MLE and the UCE game (see Figure 1). Note that the CDA and the MLE game do not necessarily require keyed hash functions but in most constructions explicitly make only keyed hash queries by embedding the public key (for CDA) and the public parameter (for MLE) respectively in every hash query.[6] For the chosen distribution attack (CDA) game [3], which captures the security of deterministic PKE schemes, the only assumption is that the public-key cannot be guessed. For the adaptive version of the CDA game one needs the additional assumption that the PKE scheme does not leak the public-key within its ciphertexts. We call the corresponding property PK-EXT (short for public key extractability) and introduce it in the full version [27]. For message-locked encryption (MLE) [7] one needs to assume that the public parameter $P$ cannot be guessed. Finally, UCE is stated directly for keyed-hash functions that is, here one needs to assume that the hash-key cannot be guessed. Note that this shows that HMAC is UCE-secure when assuming idealized compression functions which solves an open problem in [6]. We give introductions to the various notions, as well as, formal statements listing under which assumptions Theorem 2 applies to CDA, MLE and UCE in the full version of this paper [27].

**Acknowledgments.** I thank the anonymous reviewers for valuable comments. In particular, I would like to thank Paul Baecher, Christina Brzuska, Özgür Dagdelen, Pooya Farshim, Marc Fischlin, Tommaso Gagliardoni, Giorgia Azzurra Marson, and Cristina Onete for many fruitful discussions and support throughout the various stages of this work. This work was supported by CASED (`www.cased.de`).

## References

1. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to NIST (Round 3) (2010), `http://131002.net/blake/blake.pdf`

[6] For CDA, consider schemes Encrypt-With-Hash [3] and Randomized-Encrypt-With-Hash [4]. For MLE consider the convergent encryption (CE) scheme [7,19].

2. Baecher, P., Brzuska, C., Mittelbach, A.: Reset indifferentiability and its consequences. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part I. LNCS, vol. 8269, pp. 154–173. Springer, Heidelberg (2013)
3. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 535–552. Springer, Heidelberg (2007)
4. Bellare, M., Brakerski, Z., Naor, M., Ristenpart, T., Segev, G., Shacham, H., Yilek, S.: Hedged public-key encryption: How to protect against bad randomness. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 232–249. Springer, Heidelberg (2009)
5. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
6. Bellare, M., Hoang, V.T., Keelveedhi, S.: Instantiating random oracles via UCEs. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 398–415. Springer, Heidelberg (2013)
7. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Johansson and Nguyen [22], pp. 296–312
8. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Ashby, V. (ed.) ACM CCS 1993, pp. 62–73. ACM Press (November 1993)
9. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006)
10. Bennett, C.H., Gill, J.: Relative to a random oracle A, $P^A \neq NP^A \neq coNP^A$ with probability 1. SIAM Journal on Computing 10(1), 96–113 (1981)
11. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The keccak SHA-3 submission. Submission to NIST, Round 3 (2011), `http://keccak.noekeon.org/Keccak-submission-3.pdf`
12. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Cryptographic sponge functions (2011)
13. Bhattacharyya, R., Mandal, A., Nandi, M.: Indifferentiability characterization of hash functions and optimal bounds of popular domain extensions. In: Roy, B., Sendrier, N. (eds.) INDOCRYPT 2009. LNCS, vol. 5922, pp. 199–218. Springer, Heidelberg (2009)
14. Brassard, G. (ed.): CRYPTO 1989. LNCS, vol. 435. Springer, Heidelberg (1990)
15. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård revisited: How to construct a hash function. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 430–448. Springer, Heidelberg (2005)
16. Damgård, I.: A design principle for hash functions. In: Brassard [14], pp. 416–427
17. Demay, G., Gazi, P., Hirt, M., Maurer, U.: Resource-restricted indifferentiability. In: Johansson and Nguyen [22], pp. 664-683
18. Dodis, Y., Ristenpart, T., Shrimpton, T.: Salvaging Merkle-Damgård for practical applications. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 371–388. Springer, Heidelberg (2009)
19. Douceur, J.R., Adya, A., Bolosky, W.J., Simon, D., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: ICDCS, pp. 617–624 (2002)
20. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The skein hash function family. Submission to NIST (Round 3) (2010), `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`

21. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grstl – a SHA-3 candidate. Submission to NIST (Round 3) (2011), http://www.groestl.info/Groestl.pdf
22. Johansson, T., Nguyen, P.Q. (eds.): EUROCRYPT 2013. LNCS, vol. 7881. Springer, Heidelberg (2013)
23. Liskov, M.: Constructing an ideal hash function from weak ideal compression functions. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 358–375. Springer, Heidelberg (2007)
24. Luykx, A., Andreeva, E., Mennink, B., Preneel, B.: Impossibility results for indifferentiability with resets. Cryptology ePrint Archive, Report 2012/644 (2012), http://eprint.iacr.org/2012/644
25. Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (2004)
26. Merkle, R.C.: One way hash functions and DES. In: Brassard [14], pp. 428–446
27. Mittelbach, A.: Salvaging indifferentiability in a multi-stage setting. Cryptology ePrint Archive, Report 2013/286 (2013), http://eprint.iacr.org/2013/286
28. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of indifferentiability and universal composability. Cryptology ePrint Archive, Report 2011/339 (2011), http://eprint.iacr.org/2011/339
29. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of the indifferentiability framework. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011)
30. Rivest, R.: The MD5 Message-Digest Algorithm. RFC 1321 (Informational) (April 1992), http://www.ietf.org/rfc/rfc1321.txt (updated by RFC 6151)
31. Wu, H.: The hash function JH. Submission to NIST (round 3) (2011), http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf