Salvaging Indifferentiability in a Multi-stage Setting

Arno Mittelbach

Darmstadt University of Technology, Germany
www.cryptoplexity.de
arno.mittelbach@cased.de

January 24, 2014

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 that we call *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 of 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.

Contents

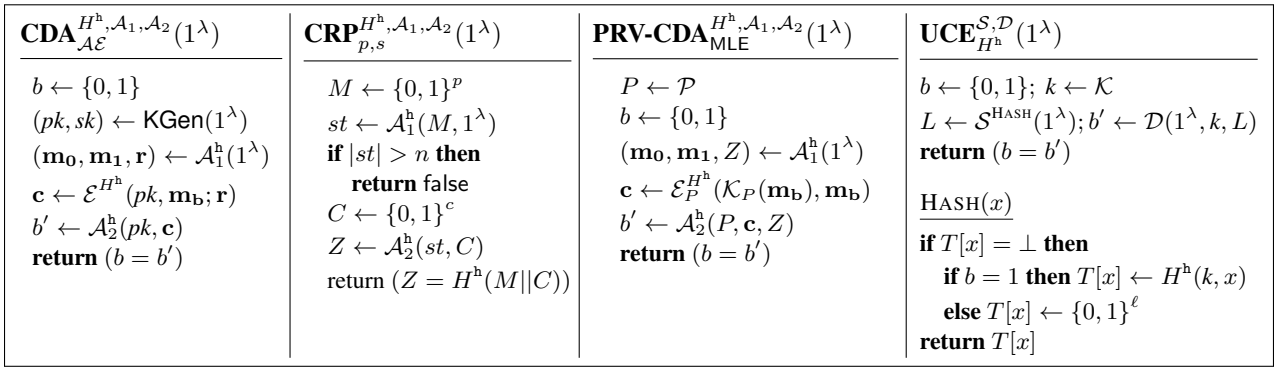

Figure 1: Security Games. From left to right: the chosen distribution attack (CDA) game [BBN+09] capturing security in deterministic encryption schemes [BBO07], the proof-of-storage challenge-response game (CRP) due to Ristenpart et al. [RSS11b] given as counter-example of the general applicability of the indifferentiability composition theorem, message locked encryption (MLE) [BKR13], and universal computational extractors (UCE) [BHK13] a standard model security assumption on hash-functions.

1 Introduction

The notion of indifferentiability, introduced by Maurer, Renner and Holenstein (MRH) [MRH04] 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 [Mer89, Dam89, Riv92, Lis06, BDPA11a]. The MRH 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) [BR93] 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) [RSS11b] 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 $H^{\mathrm{h}}(M\|\cdot)$. We refer to [RSS11b] for a detailed discussion.

To circumvent the problem of composition in multi-stage settings, RSS propose a stronger form of indifferentiability called *reset indifferentiability* [RSS11b], which intuitively states that simulators must be stateless and pseudo-deterministic [BBM13]. While this notion allows composition in any setting, no domain extender can fulfill this stronger form of indifferentiability [DGHM13, LAMP12, BBM13]. Demay et al. [DGHM13] 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 [BBN+09], depicted in Figure 1. CDA captures a security notion for deterministic public-key encryption schemes [BBO07], 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

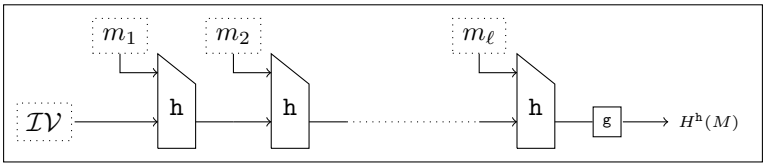

Figure 2: Merkle-Damgård Construction

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 [DRS09], 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 [CDMP05] (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 in differentiability 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 schematic of a Merkle-Damgård construction in Figure 2 (the final g-node is an efficient transformation such as the projection to the first half of the state bits, as in chop-MD) 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 argue about such problematic adversarial $\mathtt{h}$-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 UNSPLITTABLE (Definition 4.1). If a game is UNSPLITTABILITY 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 heavily rely 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 [BG81]. 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 [BCK96]) 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 [RSS11b]). Similarly, we show UNSPLITTABILITY for message locked encryption (MLE), a security definition for primitives that allow for secure deduplication [BKR13]. 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 [Lis06] that process the input message twice for computing the final hash value. Finally, we resolve an open problem from [BHK13]. 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 [BHK13]. 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 [CDMP05]. 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 ℓ 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.

In this paper we consider random oracles or ideal functions by which we mean functions that provide a uniformly chosen random output (within the specified range) on every new query.

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 [Mer89, Dam89, Riv92, Lis06, AHMP10, GKM+11, Wu11, BDPA11a, FLS+10], 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 [BDPA11b] (the design principle behind SHA-3, aka. Keccak [BDPA11a]) iterates a permutation instead of a compression function. We discuss, how this fits into our model in Appendix B.6.

Compression functions in practical constructions are often built from keyed permutations, for example, via the Davies-Meyer (DM) construction [Win83]. We chose to state our results in the ideal compression function model— where function $\mathtt{h}$ is assumed to be a fixed-length random oracle—, instead of the ideal cipher model together with a construction such as DM, as we believe it greatly improves readability as only one instead of two constructions needs to be analyzed. We note that we expect our results to hold if the compression function is instantiated via DM and an ideal cipher. We give a brief discussion in Appendix E.

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 [CDMP05].

Definition 2.1. *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 $(t_{\mathcal{D}}, t_{\mathcal{S}}, q, \epsilon)$ indifferentiable from a random oracle $\mathcal{R}$ if there exists an efficient simulator $\mathcal{S}^{\mathcal{R}}$ such that for any distinguisher $\mathcal{D}$ it holds that*

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

where the simulator runs in time at most $t_{\mathcal{S}}$, and the distinguisher runs in time at most $t_{\mathcal{D}}$ and makes at most q queries. We say $H^{\mathtt{h}}$ is (computationally) indifferentiable from $\mathcal{R}$ if ϵ is a negligible function in the security parameter λ (for polynomially bounded $t_{\mathcal{D}}$ and $t_{\mathcal{S}}$).

The advantage of a distinguisher $\mathcal{D}$ with respect to a simulator $\mathcal{S}$ in the indifferentiability game is defined as

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

We sometimes speak of the *real world* when meaning that the distinguisher is connected to hash function $H^{\mathtt{h}}$ and underlying function $\mathtt{h}$ and of the *ideal world* when it is talking to random oracle $\mathcal{R}$ and simulator $\mathcal{S}^{\mathcal{R}}$.

GAME PLAYING. We use the game-playing technique [BR06, RSS11b] and present here a brief overview of the notation used. A self-contained introduction is given in Appendix A.

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 [RSS11b]. 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. [BMN09] 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

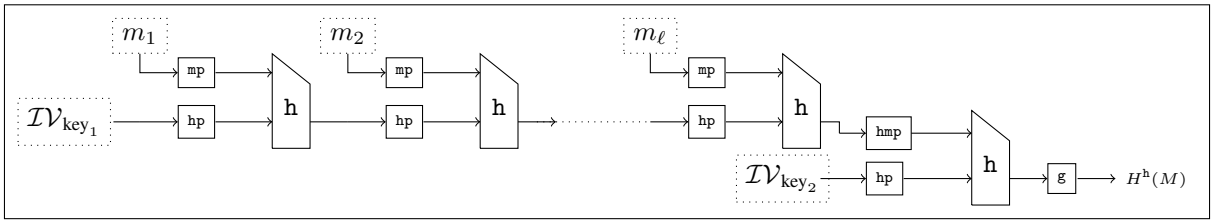

Figure 3: Execution graph for NMAC for message $m_1 \| \ldots \| m_\ell := M$. Value $\mathcal{IV}_{\text{key}_1}$ is an initialization vector representing the first key in the NMAC-construction. Value $\mathcal{IV}_{\text{key}_2}$ is a constant representing the second key. The difference between initialization 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.

types of constructions, ranging from the plain Merkle-Damgård over variants such as chop-MD or Sponge to more complex constructions such as NMAC, HMAC [BCK96] or even hash trees. In Appendix B.5 we give an overview over several hash constructions that are captured by our model.

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 3 presents the *execution graph* for a message $M := m_1 \| \ldots \| m_\ell$ for the NMAC construction [BCK96] (further examples are given in Appendix B.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 3). 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. Assume that $\mathfrak{eg}$ is the execution graph for a message $M \in \{0,1\}^*$. Then we can formalize the computation of hash value $H^{\mathtt{h}}(M)$ with underlying function $\mathtt{h}$ by a deterministic algorithm $\mathtt{EVAL}^{\mathtt{h}}(\mathfrak{eg})$ which repeats the following steps: search for a node with no input edges or where all input edges are labeled. Compute the corresponding function (if it is an $\mathtt{h}$-node, call the provided $\mathtt{h}$-oracle), remove the node and label all outgoing edges with the resulting value. The label of the single unbound outgoing edge of the $\mathtt{g}$-node is the resulting hash value.

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 [Dam89, Mer89]) 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).[1] 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 for execution graphs in Appendix B.1 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

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

contains exactly one message-node. All outgoing edges must again be connected to a node, except for the single outgoing edge of the single g-node. An h-node always has two incoming edges one from an hp-node and one from either an mp or an hmp-node. Message nodes can be connected to mp-nodes. The outbound edges from h can be connected to either hp or hmp-nodes.[2] 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 g-node. Hence, they contain exactly one unbound outgoing edge from an 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 h-node and removing every node that can be reached via directed path from that h-node and then remove all unconnected components that do not have a directed path to the chosen h-node.

We define EVAL to be a generic, deterministic algorithm evaluating execution graphs relative to an oracle h. We here present a slightly simplified, intuitive version of EVAL and give the complete version along with its pseudo-code in Appendix B.1. Let $\mathfrak{eg}$ be a valid execution graph for some message $M \in \{0,1\}^*$. To evaluate $\mathfrak{eg}$ relative to oracle h, algorithm $\texttt{EVAL}^{\texttt{h}}(\mathfrak{eg})$ 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 $\texttt{EVAL}^{\texttt{h}}(\mathfrak{eg})$ 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 $\texttt{EVAL}^{\texttt{h}}(\mathfrak{pg})$, likewise, computes the partial graph outputting the result of the final h-node. We denote by $\texttt{g}(\mathfrak{pg})$ the corresponding execution graph where the single outbound h-edge of $\mathfrak{pg}$ is connected to a 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 of the preprocessing functions, the padding function and the final transformation $\texttt{g}(\cdot)$. Furthermore, we require (efficient) algorithms that construct execution graphs as well as parse an execution graph to recover the corresponding message.

Definition 3.1. *Let $\mathcal{IV} \subset \{0,1\}^*$ be a set of named initialization vectors and $|\mathcal{IV}|$ be polynomial in the security parameter λ. We say $H^{\texttt{h}}_{\texttt{g},\texttt{mp},\texttt{hp},\texttt{hmp},\texttt{pad}} : \{0,1\}^* \to \{0,1\}^n$ is an iterative hash function if there exist deterministic and efficient algorithms* construct *and* extract *as follows:*

construct**:** *On input $M \in \{0,1\}^*$ deterministic algorithm* construct *outputs a valid execution graph containing one message-node for every block in $m_1 \| \ldots \| m_\ell := \texttt{pad}(M)$. For all messages $M \in \{0,1\}^*$ it holds that $H^{\texttt{h}}_{\texttt{g},\texttt{mp},\texttt{hp},\texttt{hmp},\texttt{pad}}(M) = \texttt{EVAL}^{\texttt{h}}(\texttt{construct}(M))$. For any two $M, M' \in \{0,1\}^*$ with $|M| = |M'|$ it holds that graphs $\texttt{construct}(M)$ and $\texttt{construct}(M')$ are identical but for labels of message-nodes.*[3]

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

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

We provide a detailed description of valid execution graphs, extensions to the model to, for example, cover keyed hash constructions, as well as several examples of hash constructions that are covered by Definition 3.1 in Appendix B. Note that neither construct nor extract gets access to the underlying function h. Also note that by definition of algorithm extract there cannot be two distinct valid execution graphs for the same message M; if $\texttt{extract}(\mathfrak{pg}) = M$ then $\mathfrak{pg}$ or $\texttt{g}(\mathfrak{pg})$ is identical to $\texttt{construct}(M)$.

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

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

3.1 Important $\mathtt{h}$-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 $\mathtt{pg}$ can be completed to a valid execution graph, that is, $\mathtt{g}(\mathtt{pg})$ is a valid execution graph. For a visualization of the query types see Figure 4.

Definition 3.2. *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,

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

Let $\mathtt{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i]$ denote the set of partial graphs such that for all $\mathtt{pg} \in \mathtt{pg}[\mathtt{h}, Q_{|_{1,\ldots,i}}, q_i]$ it holds that all $\mathtt{h}$ queries occurring during the computation of $\mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$ are in $Q_{|_{1,\ldots,i}}$ and that the final $\mathtt{h}$-query equals q_i.[4] We define the predicate $\mathsf{result}^Q(m_i, x_i)$ to be true if, and only if,

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

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

Let us rephrase the concept of chained queries. Query q_i is a chained query if either q_i is an initial query, or there exists a query q_j that came before q_i which was a chained query and for which $\mathtt{hp}(h(q_j)) = \mathtt{hp}(h(m_j, x_j)) = x_i$. Thus, if $\mathtt{hp}(\cdot)$ is the identity function, as it is for example in chop-MD, or NMAC, we require that value x_i was generated by an h-query. When later considering keyed hash functions we also need to relax the requirements of initial queries as depending on the construction any query can be an initial query (it is just a matter of how the key is chosen). We elaborate on keyed hash constructions in Appendix B.2. Also, let us stress, that the predicates hold, or do not hold, relative to the previous queries given by sequence Q and are not affected by later queries.

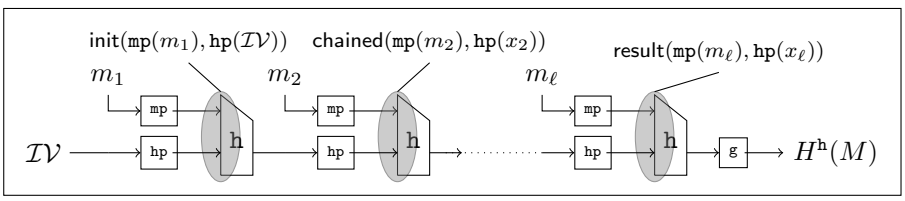

Figure 4: Denoting queries in the Merkle-Damgård construction where value x_2 is computed as $x_2 := \mathtt{h}(\mathtt{mp}(m_1), \mathtt{hp}(\mathcal{IV}))$ and value x_l is computed recursively as $x_\ell := \mathtt{h}(\mathtt{mp}(m_\ell), \mathtt{hp}(x_{\ell-1}))$.

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

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

the final g-node. As we model the underlying function as ideal, an h-evaluation has s bits of min-entropy which are, so to speak, send down the network to the final g-node. We give the proof in Appendix B.4.1 together with a strengthened version of this lemma and a variant which considers the case where the adversary in addition gets access to $H^{\mathtt{h}}$.

Lemma 3.3. *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 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*

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

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 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 give the proof in Appendix B.4.2.

Lemma 3.4. *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*

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

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 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 h of iterative hash function $H^{\mathtt{h}}$ during game G. Recall that, as do Ristenpart et al. [RSS11b], 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 h in our case).

Definition 3.5. *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. Similarly, we drop the superscript identifying oracle h exposed by the adversarial interface of functionality $H^{\mathtt{h}}$ if clear from context.

Definition 3.6. *We define 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.*

The sequence of h-queries made by the i-th adversarial procedure AdvSeq_i is defined as:

Definition 3.7. *By* qry_i *we denote the sequence of queries to* $\mathtt{h}$ *by procedure* 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 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 3.8. *By* $\mathsf{qry}_{<i}$ *we denote the sequence of queries to* $\mathtt{h}$ *before the execution of procedure* 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,*

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

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

Definition 3.9. *By* $\mathsf{qry}_{<\mathcal{A}_i,j}$ *we denote the sequence of queries to procedure* $\mathtt{h}$ *by the* i*-th adversarial procedure* AdvSeq_i *up-to query* $\mathsf{qry}_{<i,j}$*. Formally,*

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

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 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 $\mathtt{h}(\cdot,\cdot)$ badResult if it is a result query (cp. Definition 3.2) 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 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 3.10. *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* AdvSeq_i*. Then query* (m, x) *is called* $\mathsf{badResult}^{\mathcal{A}_i}(\mathsf{qry}_{i,j})$ *if, and only if:*

$$\mathsf{result}^{\mathsf{qry}_{<i,j}}(m, x) \quad \textit{and} \quad \neg\mathsf{chained}^{\mathsf{qry}_{<\mathcal{A}_i,j}}(m, x)$$

4 Unsplittable Multi-stage Games

The formalization of iterative hash functions together with the various definitions 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 (cp. Definition 3.10) 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 in Section 4.1 give the accompanying composition theorem. This informally states that if a game is UNSPLITTABLE for an indifferentiable hash construction $H^{\mathtt{h}}$, then security proofs in the random oracle carry over if the random oracle is implemented by that particular hash function.

Definition 4.1. *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*

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

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:

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

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 together with its proof in Appendix C (the theorem appears on page 42). We here only present a much shortened proof sketch.

Theorem 4.2 (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*

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

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 3.10) 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 [RSS11a] (Theorem 4 in the proceedings version [RSS11b]).

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 for any efficient distinguisher $\mathcal{D}$

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

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 5. If a query corresponds to a result query (cp. Definition 3.2) it ensures to be compatible with the random oracle by picking a value from the preimage of $\mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathfrak{pg})))$ uniformly at random (see line 7), where $\mathfrak{pg}$ 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 8). The full construction is given as Construction C.1 in Appendix C.1. One of the challenges is to argue that the so constructed simulator is indeed a good indifferentiability simulator. Lemma C.2 establishes that, indeed, if the hash construction is indifferentiable, then simulator $\mathcal{S}_*$ is a good indifferentiability simulator, and, thus

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

In a next step we derandomize simulator $\mathcal{S}_*$ using the random oracle and a derandomization technique by Bennet and Gill [BG81]. This is covered in detail in Lemma C.3 and yields simulator $\mathcal{S}_d$. The derandomization ensures that

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

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

Figure 5: Simulator $\mathcal{S}_*$ for proof of Theorem 4.2. For a detailed description see Appendix C.1 and Construction C.1. The simulator maintains a list of partial graphs that can be constructed from the sequence of queries. On a new query (m,x) the simulator creates a temporary set $\mathcal{T}$. If the query is an initial query it constructs the corresponding partial graph for it and adds it to $\mathcal{T}$. Furthermore, it tries all existing partial graphs, if they can be extended by the current query. A query is answered either by a randomly chosen value, or in case a valid execution graph was constructed for it by sampling a value uniformly at random from $\mathsf{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg}))$.

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 3.2) 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

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

$\Diamond$

5 Applications

In the following we turn to the task of proving UNSPLITTABILITY for the various multi-stage games from the introduction: the chosen distribution attack (CDA) game (Section 5.2), the message-locked encryption (MLE) game (Section 5.4), the universal computational extractor (UCE) (Section 5.5) as well as the RSS proof-of-storage game (Section 5.6). While for the RSS proof-of-storage game we will give a direct proof (see Section 5.6) we prove the results for CDA, MLE and UCE via a meta result on games using keyed hash functions.

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.

The next lemma captures 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, without changing the outcome of the game (or rather its distribution over the choice of ideal functionality $\mathtt{h}$).

Lemma 5.1. *Let game $G \in \mathcal{LG}$ be any functionality respecting game and $H^{\mathtt{h}}$ an iterative hash function with ideal function $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$. Fix random coins r and adversary $\mathcal{A}_1, \ldots, \mathcal{A}_m$. Then it holds for every value y that*

$$\Pr_{\mathtt{h}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r) \Rightarrow y \right] = \Pr_{\mathtt{h},\mathtt{g}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1^{\mathtt{h}'},\ldots,\mathcal{A}_m^{\mathtt{h}'}}(r) \Rightarrow y \right]$$

where the adversaries on the right side get access to function $\mathtt{h}'$ instead of function $\mathtt{h}$ where $\mathtt{h}'$ is defined as

$$\mathtt{h}'(m,x) := \begin{cases} \mathtt{h}(m,x) & \text{if } (m,x) \in \mathsf{qry}^{\mathtt{h}}\left[G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r) \right] \\ \mathtt{g}(m,x) & \text{else} \end{cases}$$

for an independent fixed-length random oracle $\mathtt{g}$ and the probability is over the choice of $\mathtt{h}$ in the first case and $\mathtt{h}$ and $\mathtt{g}$ in the second.

Proof. The proof is readily established by noting that if $\mathtt{h}$ is not queried on a value (m,x) then it has min-entropy s bits and the simulation by $\mathtt{h}'$ is perfect. □

In other words, if $\mathtt{h}$-queries that are not *G-relevant* are answered not by $\mathtt{h}$ but with an independently chosen random function it is sufficient that they are answered consistently over the various adversarial procedures for the game not to change.

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 (see Appendix B.2 for how keyed hash constructions are captured by our framework for iterative hash functions).

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 [CDMP05] 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 5.2. *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 (such as the UCE game; see Figure 1), 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. We see that we can recast the CDA and the MLE game in this way.

Definition 5.3. *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*. We consider keyed-hash-games where only the last stage adversary gets to see the hash key (or keys) used by the game while all previous adversarial stages do not. As we will see this exactly matches the setup of the chosen distribution attack (CDA) game (Section 5.2), the message-locked encryption (MLE) game (Section 5.4) as well as the universal computational extractor (UCE) (Section 5.5). The result is given by the following theorem.

Theorem 5.4. *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 $H^{\mathtt{h}}$.*

In a first step to prove Theorem 5.4 we give a simplified version of it. Here the setup is a two-staged keyed-hash game but it is established that the first stage adversary does not make G-relevant queries with overwhelming probability. In this case, we show that game G is UNSPLITTABLE for any key-prefixed iterative hash-function (see Definition 5.2).

Lemma 5.5. *Let $G \in \mathcal{LG}$ be a two-stage keyed-hash game where adversarial procedures $\mathcal{A}_1$ and $\mathcal{A}_2$ are called exactly once and in this order. Let $H^{\mathtt{h}}$ be a key-prefixed iterative hash-function and let $\mathtt{h} : \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^s$ be an ideal function. If for every efficient adversary $\mathcal{A}_1$ the probability of making G-relevant queries during $G^{H^{\mathtt{h}}, \mathcal{A}_1, \mathcal{A}_2}$ is negligible and adversary $\mathcal{A}_2$ gets $\mathcal{K}_G[H^{\mathtt{h}}, r]$ as part of its input then G is* UNSPLITTABLE *for $H^{\mathtt{h}}$.*

Proof. We construct adversary $\mathcal{A}_1^*$ to run $\mathcal{A}_1$ and answer any $\mathtt{h}$-query with a value drawn uniformly at random from $\{0,1\}^s$. Adversary $\mathcal{A}_1^*$ outputs whatever $\mathcal{A}_1$ outputs. Note that $\mathcal{A}_1^*$ does not use its $\mathtt{h}$-oracle. Similarly, we construct adversary $\mathcal{A}_2^*$ to run $\mathcal{A}_2$ and output whatever $\mathcal{A}_2$ outputs. Adversary $\mathcal{A}_2^*$ keeps an internal list of all $\mathtt{h}$-queries by $\mathcal{A}_2$ and constructs potential partial graphs (this can for example be done similar to the extractor from Lemma 3.4). On receiving query (m, x) it checks if all initial queries in the corresponding partial graph are correctly keyed with a key in $\mathcal{K}$ (note that $\mathcal{A}_2^*$ gets set $\mathcal{K}$ as input). If this is the case, it forwards the query to its $\mathtt{h}$ oracle. Otherwise it simply returns a random value in $\{0,1\}^s$.

By construction, it follows that any G-relevant query made by $\mathcal{A}_2$ is forwarded to $\mathtt{h}$ by $\mathcal{A}_2^*$. Furthermore, since adversary $\mathcal{A}_1$ does not make G-relevant queries (with overwhelming probability) we have that shared queries by $\mathcal{A}_1$ and $\mathcal{A}_2$ will not be G-relevant and will be answered by $\mathcal{A}_1^*$ (resp. $\mathcal{A}_2^*$) with a randomly chosen value in $\{0,1\}^s$.

It remains to ensure that shared queries by $\mathcal{A}_1$ and $\mathcal{A}_2$ are answered consistently. For this we derandomize the adversaries using a similar approach as in Lemma C.2. Let $q_{\mathtt{h}}$ denote an upper bound on the number of $\mathtt{h}$ queries by adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$. To answer query (m, x) adversary $\mathcal{A}_i^*$ will compute

$$r \leftarrow \mathtt{h}(\langle 1 \rangle, 0^k) \oplus \ldots \oplus \mathtt{h}(\langle 2q_{\mathtt{h}} + 1 \rangle, 0^k)$$
$$y \leftarrow \mathtt{h}(m, x) \oplus r$$

and return y. Because of the restrictions on the number of queries, this is a perfect simulation of generating uniformly random values for $\mathcal{A}_1$ and $\mathcal{A}_2$. Now, the same query by $\mathcal{A}_1$ and $\mathcal{A}_2$ to their oracles is answered consistently by $\mathcal{A}_1^*$ and $\mathcal{A}_2^*$. This is true unless $\mathcal{A}_2^*$ answers the query using its $\mathtt{h}$ oracle since the corresponding partial graph has correct initial queries (correct relative to $\mathcal{K}$). By the strengthened missing link lemma (Lemma B.2) the probability that $\mathcal{A}_1$ makes such a query is, however, negligible.

The proof follows with Lemma 5.1. □

Remark. Note that the lemma can be straightforwardly adapted to m adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_m$ that are each called once and in this order and where the restrictions for $\mathcal{A}_1$ apply to all adversaries except for the last stage $\mathcal{A}_m$ which takes the role of $\mathcal{A}_2$ in the lemma. In the proof simply apply the same steps as for adversary $\mathcal{A}_1$ to all but the last adversary which also here plays the role of $\mathcal{A}_2$. In the upcoming discussion we stick to the level of only two adversaries to simplify notation.

To make use of the just proven lemma, we need to give sufficient conditions under which an adversarial procedure does not make any G-relevant queries. In the following lemma we consider games where the first

adversarial procedure is not given access to the key used by the game to make keyed hash queries. This, we formalize by requiring that no extractor, given the view of $\mathcal{A}_1$ can output a key $\kappa \in \mathcal{K}_G[H^{\mathtt{h}}, r]$ where $\mathcal{K}_G[H^{\mathtt{h}}, r]$ (see Definition 5.3 of keyed hash games) denotes the hash keys used by game G with hash function $H^{\mathtt{h}}$ when run on random coins r. The view of adversary $\mathcal{A}_1$ is denoted by $\mathsf{View}[\mathcal{A}_1; H^{\mathtt{h}}, r]$ and contains the random coins of $\mathcal{A}_1$ together with its input and answers to any of its oracle queries when game G is run with coins r and function $H^{\mathtt{h}}$.

Lemma 5.6. *Let $G \in \mathcal{LG}$ be a keyed-hash game such that adversarial procedure $\mathcal{A}_1$ is the first adversarial procedure called by G and is called only once. 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 $\mathsf{View}[\mathcal{A}_1; H^{\mathtt{h}}, r]$ the view of adversary $\mathcal{A}_1$, i.e., the random coins of $\mathcal{A}_1$ together with its input and answers to any of its oracle queries.*

If for every efficient extractor $\mathcal{E}$ and for every efficient adversary $\mathcal{A}_1$ there exists negligible function $\mathtt{negl}$ such that

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

then $\mathcal{A}_1$ does not make G-relevant queries with overwhelming probability.

Proof. We show that if adversary $\mathcal{A}_1$ makes G-relevant queries, that we can then build a distinguisher that wins in the indifferentiability game. For this let $\mathcal{S}_*$ be the simulator constructed from $\mathcal{S}$ according to construction C.1. Then by Lemma C.2 and the fact that $H^{\mathtt{h}}$ is indifferentiable, we have that

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

We will later use the special structure from simulator $\mathcal{S}_*$ to argue that G-relevant queries only occur with negligible probability. For this we assume that the simulator aborts if one of the failure conditions B_1 to B_3 occurs (see pseudo-code of simulator on page 38 and second game hop in proof of Lemma C.2). By the second game hop in the proof of Lemma C.2 we know that these failure conditions only occur with negligible probability.

We construct a distinguisher $\mathcal{D}$ for the indifferentiability game as follows. Distinguisher $\mathcal{D}$ gets access to a functionality $\mathcal{F} := (\mathcal{F}.\text{hon}, \mathcal{F}.\text{adv})$ which is either $(H^{\mathtt{h}}, \mathtt{h})$ or $(\mathcal{R}, \mathcal{S}_*^{\mathcal{R}})$ and has to distinguish between the two settings. Distinguisher $\mathcal{D}$ runs game $G^{\mathcal{F}.\text{hon},\mathcal{A}_1^{\mathcal{F}.\text{adv}},\ldots\mathcal{A}_m^{\mathcal{F}.\text{adv}}}$. Let Q_G denote the set of queries by G to $\mathcal{F}.\text{hon}$ and let $Q_{\mathcal{A}}$ denote the queries by adversary $\mathcal{A}_1$ to $\mathcal{F}.\text{adv}$. After executing game G, distinguisher $\mathcal{D}$ tests if $\mathcal{A}_1$ made a G-relevant query. For this it computes for all messages $(\kappa, M) \in Q_G$ hash value $H^{\mathcal{F}.\text{adv}}(\kappa, M)$ and records the occurring $\mathcal{F}.\text{adv}$-queries in set Q_τ. Distinguisher $\mathcal{D}$ outputs 0 if the intersection is empty and 1 otherwise (in this case adversary $\mathcal{A}_1$ succeeded in making a G-relevant query).

Let us assume that the lemma does not hold and that adversary $\mathcal{A}_1$ makes G-relevant queries with noticeable probability ϵ. Then, when $(\mathcal{F}.\text{hon}, \mathcal{F}.\text{adv}) = (H^{\mathtt{h}}, \mathtt{h})$ distinguisher $\mathcal{D}$ will always notice when $\mathcal{A}_1$ makes a G-relevant query and will, thus, with probability ϵ output 1 and 0 otherwise. Let us now consider setting $(\mathcal{F}.\text{hon}, \mathcal{F}.\text{adv}) = (\mathcal{R}, \mathcal{S}_*^{\mathcal{R}})$. By equation (1) we have that the output distribution of $\mathcal{D}$ must be negligibly close to outputting 1 with probability ϵ. Thus, for equation (1) to hold it must be that also in this setting the intersection $Q_{\mathcal{A}} \cap Q_\tau$ is not empty with probability ϵ.

Consider the state of simulator $\mathcal{S}_*$ after the execution of $\mathcal{A}_1^{\mathcal{S}_*^{\mathcal{R}}}$. By construction, for each query $(m, x) \in Q_{\mathcal{A}}$ the simulator maintains at most three partial graphs such that the sole unbound edge is labeled with x (or rather $\mathtt{hp}^{-1}(x)$ but for simplicity we here simply assume that $\mathtt{hp}$ is the identity function). In the following we argue that neither type of query (queries for which the simulator maintains partial graphs and those for which it does not maintain partial graphs) can lead to a G-relevant query.

First we consider queries $(m, x) \in Q_{\mathcal{A}}$ for which the simulator maintains a partial graph such that the sole unbound edge is labeled with x. That the simulator contains such a partial graph means that the query is a chained query (or an initial query). However, with overwhelming probability it is chained with respect to a key not in $\mathcal{K}_G[H^{\mathtt{h}}, r]$ (that is a key, that is not used by the game), as otherwise we can build an extractor to extract the key. On the other hand, by construction, all queries in Q_τ are chained with respect to a key in $\mathcal{K}_G[H^{\mathtt{h}}, r]$. This means that the corresponding partial graph is different from any of the partial graphs maintained for queries in $Q_{\mathcal{A}}$. If two such partial graphs now have their sole unbound edges labeled with the same value, this

implies that a collision occurred on a query to $\mathcal{S}_*$. This, however violates failure condition B_1 and, thus, by the second game hop in the proof of Lemma C.2 this happens only with negligible probability.

It remains to consider queries $(m, x) \in Q_{\mathcal{A}}$ for which the simulator $\mathcal{S}_*$ does not maintain a partial graph after the execution of $\mathcal{A}_1^{\mathcal{S}_*^{\mathcal{R}}}$. For this, assume that there is $(\kappa, M) \in Q_G$ such that query (m, x) occurs during the computation of $H^{\mathcal{S}_*^{\mathcal{R}}}(\kappa, M)$ (and, thus, $(m, x) \in Q_\tau$). During the computation of $H^{\mathcal{S}_*^{\mathcal{R}}}(\kappa, M)$ query (m, x) is a chained query and thus simulator $\mathcal{S}_*$ will maintain a partial graph such that the sole unbound edge is labeled with value x. This, however, directly violates failure condition B_3 as here a query that appeared earlier in the computation could be used to extend a new partial graph. Again, by the second game hop in the proof of Lemma C.2 this happens only with negligible probability. □

Remark. Note that, again, the lemma can be straightforwardly adapted to m adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_m$ where all adversaries up to the last one share the restrictions of adversary $\mathcal{A}_1$. In this case the argument simply iterates over the adversaries proving at the i-th step that the i-th adversary cannot make G-relevant queries conditioned on that all previous adversaries do not make G-relevant queries.

We can now prove Theorem 5.4 which establishes UNSPLITTABILITY for any keyed-hash game by combining Lemmas 5.6 and 5.5.

Proof of Theorem 5.4. With Lemma 5.6 we have that adversaries $\mathcal{A}_1$ to $\mathcal{A}_{m-1}$ do not make any G-relevant queries. The result then follows with Lemma 5.5. Also see remarks after Lemmas for how to extend them to m adversaries. □

5.2 The Chosen Distribution Attack Game

In the following section we show UNSPLITTABILITY for the chosen distribution attack (CDA) security game. The CDA captures a security notion of deterministic public key encryption schemes [BBN$^+$09]. We begin by recalling the basic definitions for the non-adaptive CDA game.

PUBLIC-KEY ENCRYPTION. A public-key encryption scheme $\mathcal{AE} := (\mathsf{KGen}, \mathcal{E}, \mathcal{D})$ consists of three efficient algorithms: a key generation algorithm KGen that given the security parameter generates a keypair (pk, sk), an encryption algorithm $\mathcal{E}$ that, being given a message m, randomness r, and the public key *pk*, outputs a ciphertext c, and the decryption algorithm $\mathcal{D}$ that, given a ciphertext c and secret key *sk*, outputs a plaintext message or a distinguished symbol $\perp$.

CDA SECURITY. The CDA game (depicted in Figure 1) captures the security of public-key encryption schemes where the randomness used to encrypt may not be sufficiently random after all, i.e., it may not have sufficient min-entropy [BBN$^+$09]. For the remainder of this and the next section we denote by $\omega > 0$ the size of messages and by $\rho > 0$ the size of randomness for encryption scheme $\mathcal{E}$. In the CDA-game adversary $\mathcal{A}_1$ implements a so called (μ, ν)-$\mathtt{mmr}$-source which is a probabilistic algorithm that outputs a triplet of vectors $(\mathbf{m}_0, \mathbf{m}_1, \mathbf{r})$, each of size ν. Vectors $\mathbf{m}_0$ and $\mathbf{m}_1$ contain messages, that is, each component is of size ω and vector $\mathbf{r}$ corresponds to randomness, that is each component is of size ρ. Furthermore, to exclude trivial attacks, it is required that $(\mathbf{m}_b[i], \mathbf{r}[i]) \neq (\mathbf{m}_b[j], \mathbf{r}[j])$ for all $1 \leq i < j \leq \nu$ and all $b \in \{0, 1\}$. Finally, one requires that components have sufficient min-entropy μ independent of the random oracle, that is for all $1 \leq i \leq \nu$, all $b \in \{0, 1\}$, all $r \in \{0, 1\}^\rho$, and all $m \in \{0, 1\}^\omega$ it holds that

$$\Pr\Big[(\mathbf{m}_b[i], \mathbf{r}[i]) = (m, r) | (\mathbf{m_0}, \mathbf{m_1}, \mathbf{r}) \leftarrow \mathcal{A}_1^{\mathcal{R}}(1^\lambda), \mathcal{R}\Big] \leq 2^{-\mu} .$$

The advantage of an adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ in the CDA game where adversary $\mathcal{A}_1$ is a valid (μ, ν)-$\mathtt{mmr}$-source is given as

$$\mathsf{Adv}^{\mathrm{CDA}}_{\mathcal{AE}, H^{\mathrm{h}}}(\mathcal{A}_1, \mathcal{A}_2) := 2 \cdot \Pr\Big[\mathrm{CDA}^{H^{\mathrm{h}}, \mathcal{A}_1, \mathcal{A}_2}_{\mathcal{AE}} \Rightarrow \mathsf{true}\Big] - 1 .$$

CDA IS UNSPLITTABLE. To show that CDA is UNSPLITTABLE we use Therom 5.4 and show that the CDA game fulfills the requirements stated therein. The CDA game is a two-stage game. In order to meet the requirements of Theorem 5.4 we need to ensure that the game is keyed. For this, we need two things. 1) we require that all hash queries by the encryption scheme are keyed. For this we require that the encryption scheme's public key is used as hash key. 2) we need that the probability of guessing the public key for encryption scheme $\mathcal{AE}$ is negligible for adversary $\mathcal{A}_1$. We define the maximum public-key collision probability in the style of [BBO07]. This intuitively captures the probability of guessing a public key as generated by a PKE scheme's KGen algorithm:

$$\mathsf{maxpk}_{\mathcal{AE}} := \max_{w \in \{0,1\}^*} \Pr\left[(pk, sk) \leftarrow \mathsf{KGen}(1^\lambda) : pk = w \right] \tag{2}$$

Restricting CDA as described above and applying Theorem 5.4 we get:

Lemma 5.7. *Let $\mathcal{AE}$ be a public-key encryption scheme with negligible $\mathsf{maxpk}_{\mathcal{AE}}$. Let the encryption scheme query its hash functionality H^{h} using key pk as hash key. Then, the non-adaptive $CDA^{H^{\mathrm{h}},\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}$ game (cf. Figure 1) is* UNSPLITTABLE *for any key-prefixed iterative hash function.* □

We note that many encryption schemes proposed for deterministic encryption are of the form that they prepend the public-key to all their hash function queries. Examples include the Encrypt-And-Hash scheme as well as the Encrypt-With-Hash scheme from [BBO07] and the Randomized-Encrypt-With-Hash scheme from [BBN$^+$09]. Thus, if the hash function is instantiated with, for example, Merkle-Damgård variants such as chop-MD or prefix-free-MD or with HMAC or NMAC the above lemma applies.

5.3 The Adaptive Chosen Distribution Attack Game

In the adaptive CDA game [BBN$^+$09] (also see Figure 7) the first adversary can adaptively generate ciphertexts before it has to output the two message vectors $\mathbf{m_0}, \mathbf{m_1}$ and the randomness vector $\mathbf{r}$. For this, we give adversary $\mathcal{A}_1$ access to an oracle ENC, which allows to encrypt messages under the public key, but without having to give $\mathcal{A}_1$ access to the public key (except, of course, what is revealed by ENC-queries).

PK-EXT SECURITY. In order to prove that the adaptive CDA game is UNSPLITTABLE we need an extra assumption on the encryption scheme: namely, given the encryption of a message, it should be infeasible to extract the public key used in the encryption. Bellare et al. [BBDP01] define the notion of key indistinguishability (IK-CPA, see Figure 20) for public-key encryption schemes which intuitively captures that no adversary given an encryption can learn anything about the public key used for the encryption. The notion is defined as an indistinguishability notion, where a first-stage adversary gets two distinct public keys and outputs a message. According to some secret bit b this message is encrypted with one of the two public keys and given to a second stage adversary that has to guess b. Note that this notion cannot be fulfilled by any PKE scheme if the adversary is allowed to chose the randomness used in the encryption. Here the second-stage adversary can simply, on its own, recompute the ciphertext for both public keys and compare the outcome to its input.

We propose a weaker notion that can be met even if the adversary chooses the randomness used by the encryption scheme. We define the notion of PK-EXT (short for public-key extractability) for public-key encryption schemes. Game $\text{PK-EXT}^{\mathcal{A},H^{\mathrm{h}}}_{\mathcal{AE}}$ is shown in Figure 6. An adversary can make multiple queries to an encryption oracle and then has to output a guess for the public key that was used for the encryptions. We define the advantage of an adversary $\mathcal{A}$ by

$$\mathsf{Adv}^{\text{PK-EXT}}_{\mathcal{AE}}(\mathcal{A}) := \Pr\left[\text{PK-EXT}^{\mathcal{A}}_{\mathcal{AE}} \Rightarrow \mathsf{true}\right].$$

Note that this property is a natural strengthening of the property that public keys output by the key generation algorithm should not be guesseable. However, it is still quite a weak property as it only requires that super-logarithmically many bits of the public key have to remain hidden. In Appendix D we prove that our new notion is met by the REwH1 scheme [BBN$^+$09] if the underlying PKE scheme is IK-CPA secure. Examples of IK-CPA-secure schemes are, for example, the El Gamal or the Cramer-Shoup schemes [CS98]. We can further show that in case the adversary cannot specify the randomness, then PK-EXT is directly implied by IK-CPA.

$\text{PK-EXT}^{\mathcal{A},H^{\mathtt{h}}}_{\mathcal{AE}}$	**procedure** $\mathtt{ENC}(m, r)$
$(pk, sk) \leftarrow \mathsf{KGen}(1^\lambda)$ $pk' \leftarrow \mathcal{A}^{\mathtt{ENC},\mathtt{h}}(1^\lambda)$ return $(pk = pk')$	return $\mathcal{E}^{H^{\mathtt{h}}}(pk, m; r)$

Figure 6: Game PK-EXT

$\mathbf{aCDA}^{H^{\mathtt{h}},\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}(1^\lambda)$	**procedure** $\mathtt{ENC}(\mathbf{m_0}, \mathbf{m_1}, \mathbf{r})$
$b \leftarrow \{0,1\}$ $(pk, sk) \leftarrow \mathsf{KGen}(1^\lambda)$ $(\mathbf{m_0}, \mathbf{m_1}, \mathbf{r}) \leftarrow \mathcal{A}_1^{\mathtt{ENC},\mathtt{h}}(1^\lambda)$ $\mathbf{c} \leftarrow \mathcal{E}^{H^{\mathtt{h}}}(pk, \mathbf{m_b}; \mathbf{r})$ $b' \leftarrow \mathcal{A}_2^{\mathtt{h}}(pk, \mathbf{c})$ return $(b = b')$	return $\mathcal{E}^{H^{\mathtt{h}}}(pk, \mathbf{m_b}; \mathbf{r})$

Figure 7: The adaptive CDA game

THE ADAPTIVE CDA GAME IS UNSPLITTABLE. The proof in the adaptive setting is essentially equivalent to the proof in the non-adaptive setting: it follows with Theorem 5.4.

Lemma 5.8. *Let $\mathcal{AE}$ be a public-key encryption scheme such that for any efficient adversary $\mathcal{A}$ the advantage against public-key extractibility is negligible:*

$$\mathsf{Adv}^{\textit{PK-EXT}}_{\mathcal{AE}}(\mathcal{A}) \leq \mathtt{negl}$$

Let the encryption scheme query its hash functionality $H^{\mathtt{h}}$ using key pk as hash key. Then, the adaptive $CDA^{H^{\mathtt{h}},\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}$ game (cf. Figure 1) i is UNSPLITTABLE *for any key-prefixed iterative hash function.* □

5.4 Message Locked Encryption

Message locked encryption [BKR13] is a notion very similar to CDA, yet for the symmetric setting. It is a security notion for symmetric encryption schemes where the encryption key is derived from the to-be encrypted message. This allows for secure deduplication of data, a property useful, for example, in the cloud storage setting. Here, a storage provider wants to save storage capacity by not storing equivalent files multiple times (encrypted under different keys). If the encryption key only depends on the message (and possible public parameters), then the cloud provider can detect multiple copies of the same file and store it only once.

An MLE scheme consists of five algorithms $\mathsf{MLE} := (\mathcal{P}, \mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{T})$ where $\mathcal{K}, \mathcal{E}, \mathcal{D}$ is a symmetric-encryption scheme, $\mathcal{P}$ is a probabilistic algorithm to generate a public parameter P and $\mathcal{T}$ is a tagging algorithm (which is used for deduplication and not important for our discussion).

In Figure 8 we give the $\mathsf{IND\text{-}CDA}_{\mathsf{MLE}}$ and $\mathsf{IND\$\text{-}CDA}_{\mathsf{MLE}}$ security games on the left and the popular convergent encryption (CE) scheme [BKR13, DAB+02] on the right. In the $\mathsf{IND\text{-}CDA}_{\mathsf{MLE}}$ security game a public parameter P is generated in the first step. Then the first adversarial stage $\mathcal{A}_1$ is run (without having access to parameter P) and outputs two message vectors $\mathbf{m}_0, \mathbf{m}_1$ as well as some state Z. Similarly to CDA it is required that all entries in $\mathbf{m}_i$ are of the same length for $i \in \{0,1\}$ and vectors $\mathbf{m}_0$ and $\mathbf{m}_1$ have the same length. Further, $\mathbf{m}_j i_1 \neq \mathbf{m}_j i_2 j$ for any two distinct $i_1, i_2 \in [|\mathbf{m}_0|]$ and $j \in \{0,1\}$, that is, all values in each vector are distinct. Finally, both vectors should be unpredictable given state Z, that is each entry needs to have sufficient min-entropy conditioned on Z. According to a secret bit b every entry in $\mathbf{b}$ is then encrypted using a key constructed by algorithm $\mathcal{K}_P$ that gets public parameter P and the message to be encrypted. Then the second stage adversary gets ciphertext vector $\mathbf{c}$ and state Z and has to guess hidden bit b.

Variant $\mathsf{IND\$\text{-}CDA}_{\mathsf{MLE}}$ captures a stronger property demanding that encryptions of unpredictable messages are indistinguishable from random strings of the same length.

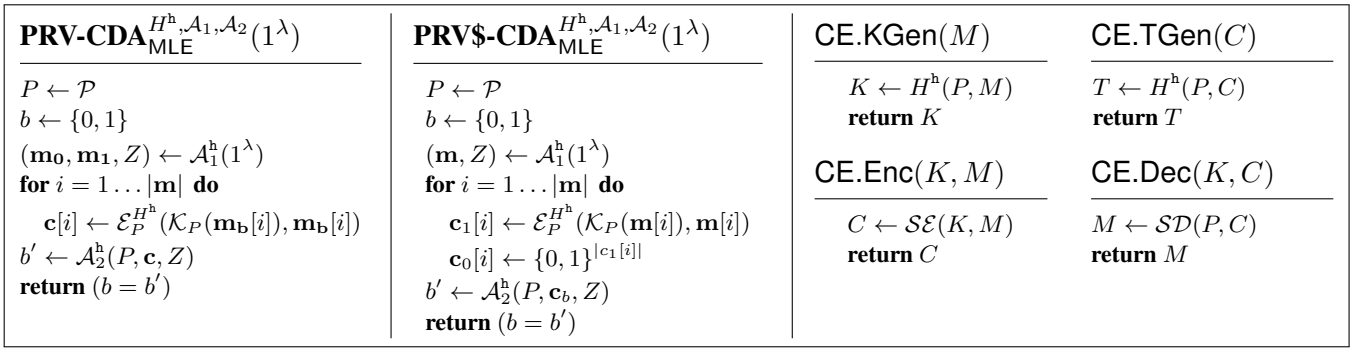

Figure 8: The PRV-CDA and PRV$-CDA MLE-security games from [BKR13] on the left. On the right the convergent encryption (CE) using a symmetric encryption scheme $\mathsf{SE} = (\mathcal{SK}, \mathcal{SE}, \mathcal{SD})$. Note that the tag generation algorithm TGen is not relevant for the security games.

Both security games are closely related to the CDA security game. One crucial difference, however, is that the two adversarial stages are able to communicate almost in the clear via state Z. In their CDA security proof for NMAC, Ristenpart et al. [RSS11b] use a strong property on the encryption scheme to make the two adversarial stages completely independent. This technique will not work for MLE, since via Z the two stages are always dependent on one another.

Using Theorem 5.4 it is, however, easily seen that both security games IND-CDA$_{\mathsf{MLE}}$ and IND$-CDA$_{\mathsf{MLE}}$ are UNSPLITTABLE for key-prefixed hash functions as long as the probability of guessing public parameter P is negligible:

Lemma 5.9. *Let $\mathsf{MLE} := (\mathcal{P}, \mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{T})$ be an MLE scheme scheme that only makes keyed queries to hash function H^{h} using parameter P as generated by algorithm $\mathcal{P}$. Then, the IND-CDA$_{\mathsf{MLE}}$ and IND$-CDA$_{\mathsf{MLE}}$ games are* UNSPLITTABLE *for any key-prefixed iterative hash function.* □

5.5 Universal Computational Extractors

Universal computational extractors (UCE) are a recently introduced standard model assumption by Bellare, Hoang and Keelveedhi [BHK13] that aims at replacing random oracles for a large class of applications. The idea is to have constructions proven as UCE-secure possibly in an idealized model (so far only random oracle constructions are known) and then to base the security of applications on the UCE assumption rather than the random oracle assumption directly.

Bellare et al. showed that the hash construction $H^{\mathcal{R}}(\kappa, m) := \mathcal{R}(k \| m)$ is UCE secure in the random oracle model, where $\mathcal{R}$ is a random oracle. They conjectured, that also HMAC is UCE secure in the idealized model, where the iterated compression function is assumed to be ideal: that is, exactly the model we are studying in this paper.

In the following we show that their conjecture was correct, that is, HMAC is UCE secure (UCE2, to be precise [BHK13]). In fact, we show a stronger statement, namely that the UCE security game depicted in Figure 9 is UNSPLITTABLE for any key-prefixed iterative hash function that is indifferentiable from a random oracle.

The UCE game (see Figure 9) is a two-stage keyed-hash game. Initially, the game chooses a hash key κ from the key space. Then the first adversary, the so-called source $\mathcal{S}$ is run and outputs some leakage L. The source has access to an oracle Hash which, according to hidden bit b returns either real hash values (under key κ) or uniformly random values. The leakage L is then given to a distinguisher $\mathcal{D}$ together with the hash key κ (note that the source did not get the hash key) and has to guess whether the source was talking to the actual hash function or not. The rule out trivial attacks, it is required that the source is unpredictable, that is, for all efficient predictors it holds that the probability of finding a HASH query by the source given leakage L is negligible.

Applying Theorem 5.4 we get:

Lemma 5.10. *Let H^{h} be a key-prefixed iterative hash-function with fixed message-rate, that is indifferentiable from a random oracle, then H^{h} is UCE2-secure.* □

$\mathbf{UCE}^{\mathcal{S},\mathcal{D}}_{H^{\mathtt{h}}}(1^\lambda)$

$b \leftarrow \{0,1\};\ \kappa \leftarrow \mathcal{K}$
$L \leftarrow \mathcal{S}^{\mathtt{h},\text{HASH}}(1^\lambda); b' \leftarrow \mathcal{D}^{\mathtt{h}}(1^\lambda, k, L)$
return $(b = b')$

HASH(x)
if $T[x] = \perp$ **then**
  if $b = 1$ **then** $T[x] \leftarrow H^{\mathtt{h}}(\kappa, x)$
  else $T[x] \leftarrow \{0,1\}^\ell$
return $T[x]$

Figure 9: The UCE-security game from [BHK13].

5.6 The Proof-Of-Storage Game and Multi-Round Hash Functions

With the next lemma we establish that the challenge-response game (cf. Figure 1) from the introduction is UNSPLITTABLE for any two-round iterative hash construction. Liskov's Zipper Hash construction [Lis06] is an example of a two-round hash function. In an r-round hash construction the entire message is processed r-times. We give an introduction to multi-round iterative hash constructions and how they are captured in our framework in Appendix B.3.

Lemma 5.11. *The proof-of-storage game* $\boldsymbol{CRP}^{H^{\mathtt{h}},\mathcal{A}_1,\mathcal{A}_2}_{p,c}$ *(cf. Figure 1) is* UNSPLITTABLE *for any r-round iterative hash construction $H^{\mathtt{h}}_r$ with $r \geq 2$.*

Proof. We show that no adversary $(\mathcal{A}_1, \mathcal{A}_2)$ has noticeable probability in winning the CRP game in case the hash functionality is instantiated with a two-round indifferentiable hash construction.

To win in the CRP game with a two-round hash function, $\mathcal{A}_2$ must compute value $H^{\mathtt{h}}_2(M||C)$ which, according to Definition 3.1, can be written as $\mathtt{g}(\mathtt{h}(m, x))$ where $\mathtt{g}$ is some transformation and (m, x) is the input to the final $\mathtt{h}$-call in the second round of the computation of $H^{\mathtt{h}}_2(M||C)$. By Lemma 3.3 we have that if only a single $\mathtt{h}$-query in the evaluation of $H^{\mathtt{h}}_2(M||C)$ is not queried, then the probability of outputting $H^{\mathtt{h}}_2(M||C)$ is at most

$$\frac{q_{\mathcal{A}_1} + q_{\mathcal{A}_2}}{2^s} + \frac{1}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))}} .$$

We now argue that adversary $\mathcal{A}_1$ cannot query $\mathtt{h}$-queries in $H^{\mathtt{h}}_2(M||C)$ that correspond to $\mathtt{h}$-nodes occurring in the second round. Challenge C is of length c bits and, thus, can be guessed by $\mathcal{A}_1$ with probability at most 2^{-c}. As C will be part of one or multiple message-block-nodes in the execution graph $\mathtt{construct}(M||C)$, we have that $\mathcal{A}_1$ is able to make all $\mathtt{h}$-queries in the first round of $\mathtt{construct}(M||C)$ with probability at most

$$\frac{q_{\mathcal{A}_1}}{2^c} .$$

Similarly, adversary $\mathcal{A}_2$ is given only n bits of information from the first round and is, thus, missing at least $p-n$ bits of message M. Thus, the probability, that it is able to make all $\mathtt{h}$-queries in the second round is upper bounded by

$$\frac{q_{\mathcal{A}_2}}{2^{p-n}} .$$

If we set adversary $(\mathcal{A}^*_1, \mathcal{A}^*_2)$ as $\mathcal{A}^*_1 = \mathcal{A}_1$ and $\mathcal{A}^*_2$ as the procedure that simply outputs a guess for value Z, then we have that bad result queries do not occur and we have that for all values y

$$\Pr\Big[\mathbf{CRP}^{H^{\mathtt{h}},\mathcal{A}_1,\mathcal{A}_2}_{p,c} \Rightarrow y\Big] \leq \Pr\Big[\mathbf{CRP}^{H^{\mathtt{h}},\mathcal{A}^*_1,\mathcal{A}^*_2}_{p,c} \Rightarrow y\Big] + \frac{q_{\mathcal{A}_1}}{2^{-c}} + \frac{q_{\mathcal{A}_2}}{2^{p-n}} + \frac{q_{\mathcal{A}_1} + q_{\mathcal{A}_2}}{2^s} + \frac{1}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))}} .$$

□

5.7 A Conjecture on Two-Stage Games and Future Work

Finally we want to present a conjecture on games consisting of exactly two stages (note that all security games in this paper are examples of a two stage game). We include this conjecture, since we believe it provides some insights into the nature of multi-stage games and multi-round hash constructions. Furthermore, proving such a result would be a huge step forward, as this would be the first truly generic positive result on indifferentiability in a multi-stage setting.

Conjecture 5.12. *Any two-stage functionality-respecting game is* UNSPLITTABLE *for any r-round iterative hash function $H_r^{\mathtt{h}}$ with $r \geq 2$.*

The idea behind Conjecture 5.12 is simple. In a two-stage game, a bad query can only be made by the second-stage adversary. Let us consider two-round hash functions. Then we can distinguish between two cases in the event that $\mathsf{bad}(m, x)$ occurs for some query (m, x) by $\mathcal{A}_2$. Let $\mathtt{pg}$ be the partial execution graph corresponding to query (m, x). Then, either (m, x) corresponds to an $\mathtt{h}$-node in $\mathtt{pg}$ in the first, or in the second round (the probability of it corresponding to two $\mathtt{h}$-nodes can be upper bounded by the probability of an $\mathtt{h}$-collision).

In the first case, the entire second round is computed by the second stage adversary $\mathcal{A}_2$ and thus it must know the entire message corresponding to the partial graph as all message-block-nodes from round 1 reappear in round 2 (see Definition B.1). In this case, however, $\mathcal{A}_2$ could have simply computed $H_2^{\mathtt{h}}(M)$ directly. For the second case a similar argument applies. Here the entire first round is computed by $\mathcal{A}_1$ which must hence know the entire message M. As bad queries can only occur with non-negligible probability if there is sufficient communication between the adversaries, adversary $\mathcal{A}_1$ could, thus, have also passed on $H^{\mathtt{h}}(M)$ instead of some intermediate value.

Acknowledgments

I thank the anonymous reviewers for valuable comments. Furthermore, I would like to thank my group members at cryptoplexity (`www.cryptoplexity.de`) for many fruitful discussions. This work was supported by CASED (`www.cased.de`).

References

[AHMP10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and Raphael C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010. (Cited on page 5.)

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, December 2001. (Cited on pages 18 and 44.)

[BBM13] Paul Baecher, Christina Brzuska, and Arno Mittelbach. Reset indifferentiability and its consequences. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 154–173. Springer, December 2013. (Cited on page 3.)

[BBN+09] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 232–249. Springer, December 2009. (Cited on pages 3, 17, 18, and 44.)

[BBO07] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 535–552. Springer, August 2007. (Cited on pages 3 and 18.)

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, August 1996. (Cited on pages 5, 7, and 34.)

[BDPA11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak SHA-3 submission. Submission to NIST (Round 3), 2011. (Cited on pages 3 and 5.)

[BDPA11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011. (Cited on pages 5 and 36.)

[BG81] C. H. Bennett and J. Gill. 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. (Cited on pages 4, 12, and 41.)

[BHK13] Mihir Bellare, Viet Tung Hoang, and Sriram Keelveedhi. Instantiating random oracles via UCEs. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 398–415. Springer, August 2013. (Cited on pages 3, 5, 20, and 21.)

[BKR13] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In Johansson and Nguyen [JN13], pages 296–312. (Cited on pages 3, 5, 19, and 20.)

[BMN09] Rishiraj Bhattacharyya, Avradip Mandal, and Mridul Nandi. Indifferentiability characterization of hash functions and optimal bounds of popular domain extensions. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT 2009*, volume 5922 of *LNCS*, pages 199–218. Springer, December 2009. (Cited on page 6.)

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Cited on page 3.)

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, May / June 2006. (Cited on pages 6 and 26.)

[Bra89] Gilles Brassard, editor. *CRYPTO'89*, volume 435 of *LNCS*. Springer, August 1989. (Cited on page 24.)

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård revisited: How to construct a hash function. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, August 2005. (Cited on pages 4, 5, 6, 14, 28, 39, and 48.)

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, August 1998. (Cited on page 18.)

[DAB+02] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002. (Cited on page 19.)

[Dam89] Ivan Damgård. A design principle for hash functions. In Brassard [Bra89], pages 416–427. (Cited on pages 3, 5, and 7.)

[DGHM13] Gregory Demay, Peter Gazi, Martin Hirt, and Ueli Maurer. Resource-restricted indifferentiability. In Johansson and Nguyen [JN13], pages 664–683. (Cited on page 3.)

[DRS09] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. Salvaging Merkle-Damgård for practical applications. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 371–388. Springer, April 2009. (Cited on pages 4 and 48.)

[FLS+10] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. Submission to NIST (Round 3), 2010. (Cited on page 5.)

[GKM+11] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schlffer, and Sren S. Thomsen. Grstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011. (Cited on page 5.)

[JN13] Thomas Johansson and Phong Q. Nguyen, editors. *EUROCRYPT 2013*, volume 7881 of *LNCS*. Springer, May 2013. (Cited on pages 23 and 24.)

[Jou04] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316. Springer, August 2004. (Cited on page 35.)

[LAMP12] Atul Luykx, Elena Andreeva, Bart Mennink, and Bart Preneel. Impossibility results for indifferentiability with resets. Cryptology ePrint Archive, Report 2012/644, 2012. `http://eprint.iacr.org/2012/644`. (Cited on page 3.)

[Lis06] Moses Liskov. Constructing an ideal hash function from weak ideal compression functions. In Eli Biham and Amr M. Youssef, editors, *SAC 2006*, volume 4356 of *LNCS*, pages 358–375. Springer, August 2006. (Cited on pages 3, 5, 21, and 28.)

[Luc04] Stefan Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253, 2004. `http://eprint.iacr.org/2004/253`. (Cited on pages 27, 35, and 36.)

[Mer89] Ralph C. Merkle. One way hash functions and DES. In Brassard [Bra89], pages 428–446. (Cited on pages 3, 5, and 7.)

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, February 2004. (Cited on page 3.)

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992. Updated by RFC 6151. (Cited on pages 3 and 5.)

[RSS11a] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of indifferentiability and universal composability. Cryptology ePrint Archive, Report 2011/339, 2011. http://eprint.iacr.org/2011/339. (Cited on pages 12 and 42.)

[RSS11b] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 487–506. Springer, May 2011. (Cited on pages 3, 5, 6, 10, 12, 20, 26, 42, and 43.)

[Win83] Robert S. Winternitz. Producing a one-way hash function from DES. In David Chaum, editor, *CRYPTO'83*, pages 203–207. Plenum Press, New York, USA, 1983. (Cited on pages 6 and 48.)

[Wu11] Hongjun Wu. The hash function JH. Submission to NIST (round 3), 2011. (Cited on page 5.)

A Game Playing

For the discussion in this paper we use the game playing technique as described in [BR06, RSS11b]. Games consist of procedures which in turn consist of a sequence of statements together with some input and zero or more outputs. Procedures can call other procedures. If procedures P_1 and P_2 have inputs and outputs that are identical in number and type, we say that they export the same interface. If a procedure P gets access to procedure $\mathcal{F}$ we denote this by adding it in superscript $P^{\mathcal{F}}$. All variables used by procedures are assumed to be of local scope. After the execution of a procedure the variable values are left as they were after the execution of the last statement. If procedures are called multiple times, this allows them to keep track of their state.

A functionality $\mathcal{F}$ is a collection of two procedures $\mathcal{F}$.hon and $\mathcal{F}$.adv, with suggestive names "honest" and "adversarial". Adversaries 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 hash functionalities which will either be instantiated with typical iterative hash constructions or with random oracles. For iterative hash constructions the adversarial interface accesses the compression function and the honest interface provides access to the complete hash function as specified, i.e., $\mathcal{F}.hon := H^{\mathtt{h}}$ and $\mathcal{F}.adv := h$. Note that access to the compression function is sufficient to compute $H^{\mathtt{h}}$. For a random oracle, on the other hand, there is no distinction between adversary access and honest access and we can assume that the adversarial interface simply forwards calls to the honest interface. As in this paper we are solely talking about iterative hash functions we will usually not write $\mathcal{F}$.hon and $\mathcal{F}$.adv, but directly refer to hash function $H^{\mathtt{h}}$ and underlying function $\mathtt{h}$, respectively.

A game G consists of a distinguished procedure called **main** (which takes no input) together with a set of procedures. A game can make use of functionality $\mathcal{F}$ and adversarial procedures $\mathcal{A}_1, \ldots, \mathcal{A}_m$ (together called "the adversary"). Adversarial procedures have access to the adversarial interface of functional procedures and, as any other procedure, can be called multiple times. We, however, restrict access to adversarial procedures to the game's **main** procedure, i.e., only it can call adversarial procedures and, in particular, adversarial procedures cannot call one another directly.

By $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ we denote a game using functionality $\mathcal{F}$ and adversary $\mathcal{A}_1, \ldots, \mathcal{A}_m$. If $\mathcal{F}'$ exports the same interface as $\mathcal{F}$, and for $1 \leq i \leq m$ adversary $\mathcal{A}'_i$ exports the same interface as $\mathcal{A}_i$, then $G^{\mathcal{F}',\mathcal{A}'_1,\ldots,\mathcal{A}'_m}$ executes the same game G with functional procedure $\mathcal{F}'$ and adversary $\mathcal{A}'_1, \ldots, \mathcal{A}'_m$. We denote by $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y$ the event that game G produces output y, that is procedure **main** returns value y. If game G uses any probabilistic procedure then $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}$ is a random variable and by $\Pr\left[\, G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y \,\right]$ we denote the probability (over the combined randomness space of the game) that it takes on value y. Sometimes we need to make the random coins r explicit and write $G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}(r)$ to denote that the game is run on random coins r.

Games are random variables over the entire random coins of the game and the adversarial procedures. For functionalities $\mathcal{F}$ and $\mathcal{F}'$ and adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_m$ and $\mathcal{A}'_1, \ldots, \mathcal{A}'_m$, we can thus consider the distance between the two random variables. Our security approach is that of concrete security, i.e., we say two games are ϵ-close if for all values y it holds that

$$\Pr\left[\, G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y \,\right] \leq \Pr\left[\, G^{\mathcal{F}',\mathcal{A}'_1,\ldots,\mathcal{A}'_m} \Rightarrow y \,\right] + \epsilon \,.$$

In asymptotic terms this means that if ϵ is negligible in the security parameter, then it follows that for all efficient distinguishers the two games are indistinguishable:

$$\left| \Pr\left[\, \mathcal{D}(G^{\mathcal{F},\mathcal{A}_1,\ldots,\mathcal{A}_m}, 1^\lambda) = 1 \,\right] - \Pr\left[\, \mathcal{D}(G^{\mathcal{F}',\mathcal{A}'_1,\ldots,\mathcal{A}'_m}, 1^\lambda) = 1 \,\right] \right| \leq \epsilon(\lambda) \,.$$

FUNCTIONALITY RESPECTING GAMES. In this paper we only consider the class of functionality-respecting games $\mathcal{LG}$ as defined by Ristenpart et al. [RSS11b]. A game is called *functionality respecting* if only adversarial procedures can call the adversarial interface of functionalities. Note that this restriction is quite natural if a game is used to specify a security goal in the random oracle model since random oracles do not provide any adversarial interface.

B Formalizing Iterative Hash Functions

B.1 Execution Graphs

In the following section we formally describe execution graphs for iterative hash functions (see Section 3). We describe the structure of the execution graph for message $m_1 \| \ldots \| m_\ell := \mathtt{pad}(M)$. An execution graph is a directed graph where nodes represent constants or functions while edges define the evaluation path. An execution graph contains exactly one unbound outgoing edge. The graph consists of the following node and edge types:

$\mathcal{IV}$-node: For every string iv $\in \mathcal{IV}$ there can exist an $\mathcal{IV}$-node with in-degree 0. Outgoing edges are of type h-edge or m-edge labeled with value iv. If the outgoing edge is of type h-edge (resp. m-edge) it must hold that iv $\in \{0,1\}^s$ (resp. iv $\in \{0,1\}^b$).

message-node: For every message block m_i (for $1 \leq i \leq \ell$) there exists a node with in-degree 0 and out-degree at least 1. Outgoing edges are of type m-edge labeled with value m_i (possibly prefixed or post-fixed with the message block counter).

mp-node: A mp-node has in-degree 1 which takes an m-edge and out-degree 1. The outgoing edge is of type mp-edge.

hp-node: A hp-node has in-degree 1 taking an h-edge and out-degree 1. The outgoing edge is of type hp-edge.

hmp-node: A hmp-node has in-degree 1 taking an h-edge and out-degree 1. The outgoing edge is of type mp-edge. We add the additional restriction that ingoing h-edges may not come from $\mathcal{IV}$-nodes.

h-nodes: An h-node has in-degree 2, an mp-edge and a hp-edge, and has out-degree at least 1. Outgoing edges are of type h-edge.

g-node: There exists a single g-node with in-degree 1, taking an h-edge and out-degree 1. The outgoing edge is not connected to a node.

We call $\mathcal{IV}$ and message-nodes *value-nodes* and all other node types *function-nodes*. All outgoing edges must be connected to a node with the only exception being the outbound edge from the single g-node.

A *valid execution graph* is a graph that is not empty and complies with the above rules. For each message $M \in \{0,1\}^*$ there is exactly one valid execution graph.

We will also need the concept of *partial execution graphs* which is a non-empty graph that complies to the above specified rules with the only exception that it does not contain a g-node. However, it must contain exactly one unbound outgoing h-edge.

We define EVAL to be a generic, deterministic algorithm evaluating execution graphs relative to an oracle h. Let $\mathtt{pg}$ be a an execution graph for some message $M \in \{0,1\}^*$. To evaluate $\mathtt{pg}$ relative to oracle h, algorithm $\mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$ first verifies the graph structure validating ensuring it is either a valid execution graph or a partial execution graph (note that this is independent of additional restrictions put due to a concrete construction). It then performs the following steps to compute the hash value: search for a node that has no inbound edges or for which all inbound edges are labeled. If the node is a value node, then remove the node (in this case the outgoing edges are already labeled). If the node is a function node then evaluate the corresponding function using the labels from the inbound edges as input. 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 $\mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$ runs in time at most $\mathcal{O}\left(|V^2|\right)$ assuming that $\mathtt{pg}$ contains $|V|$ many nodes. Note that if $\mathtt{pg}$ is a partial execution graph then $\mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$, likewise, computes the partial graph outputting the result of the final h-node. Further, if $\mathtt{pg}$ is a partial execution graph, then we denote by $\mathtt{g}(\mathtt{pg})$ the corresponding execution graph where the single outbound h-edge of $\mathtt{pg}$ is connected to a g-node. We call this the *completed* execution graph for $\mathtt{pg}$. We give the pseudo-code of algorithm EVAL in Figure 10.

Remark. In the above model, we have defined the preprocessing nodes to have in-degree 1. For certain constructions (such as, for example, the double-pipe construction [Luc04], see Section B.5.4) this requirement needs to be relaxed. Such relaxations slightly complicate the definition of initial, chained and result queries (see Section 3.1) but do not change the presented results (in an asymptotic setting).

Algorithm: $\texttt{EVAL}^h(\texttt{pg})$

$y \leftarrow \perp$
if (pg is not correct partial graph) **then return** y
while (pg contains nodes) **do**
  foreach (node in pg) **do**
    if (node is value-node) **then** remove node from pg
    if (node is function-node $\wedge$ all inbound edges are labeled) **then**
      $y \leftarrow \mathsf{evaluate}^{\mathtt{h}}(\texttt{node})$
      label all outgoing edges of node with y
      remove node and inbound edges from pg
return y

Algorithm: $\mathsf{evaluate}^{\mathcal{O}}(\texttt{node})$

if (node is mp-node) **then**
  return mp(node.in-m)
if (node is hp-node) **then**
  return hp(node.in-h)
if (node is hmp-node) **then**
  return hmp(node.in-h)
if (node is h-node) **then**
  $m \leftarrow$ node.in-mp
  $x \leftarrow$ node.in-hp
  return $\mathcal{O}(m, x)$
if (node is g-node) **then**
  return g(node.in-h)

Figure 10: The generic evaluation algorithm $\texttt{EVAL}^h$. For the evaluation of function nodes we denote by node.in-T the label of the ingoing edge of type T.

B.2 Keyed Hash Constructions

Hash functions can be keyed, that is, hash values are computed relative to a key. Keying of hash functions can be done either by design (which we refer to as explicit) or implicitly by embedding the key in the message before computing the hash value. Example of explicitly keyed hash functions are HMAC and NMAC. Examples of implicitly keyed hash functions would be the plain Merkle-Damgård function where the key is always prepended to the message before hashing it. That is, for example, for chop-MD [CDMP05] we could have the following keyed construction.

$$\mathsf{keyed\text{-}chopMD}(\kappa, M) := \mathsf{chopMD}(\kappa \| M)$$

To capture keyed hash constructions $H^{\mathtt{h}} : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$ in our framework we need to extend Definition 3.1 and additionally define the key-space $\mathcal{K}$ and specify how keys are included in the execution graph. Algorithm construct then takes as input a message $M \in \{0,1\}^*$ and key $\kappa \in \mathcal{K}$. In execution graphs we introduce a special node type for keys called κ-node. For easier notation we reuse preprocessing functions hp and mp but we note that also new functions could be introduced.

κ-node κ-nodes have in-degree and out-degree 1. The outgoing edges are of type m-edge or h-edge and the i-th κ-node is labeled by κ.

Furthermore, we require that algorithm extract has to extract not only the message but also the corresponding key(s).

INITIAL QUERIES. Depending on the iterative hash construction we also have to adapt the definition of initial queries; Definition 3.2. So far, we defined initial queries with respect to set $\mathcal{IV}$. In keyed hash constructions a κ-node might be used instead of an $\mathcal{IV}$-node (this is, for example, the case with NMAC). For such hash constructions we relax the definition of initial queries and simply assume that any query is potentially an initial query. Note that for NMAC this is indeed the case and that it is just a matter of key choice.

B.3 Multi-Round Iterative Hash Functions

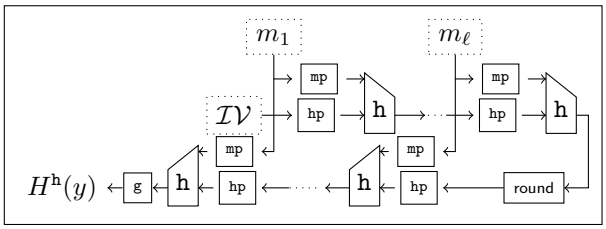

Figure 11: Zipper Hash in accordance to Definition B.1

Most hash functions only make a single pass over the message to compute the hash value. Multiple message-passes (or rounds, as we call them) may, however, lead to a *stronger* hash function. A good example of such a multi-round hash function is Liskov's Zipper Hash [Lis06] and we have depicted the corresponding execution graph in Figure 11. Zipper Hash can be regarded as a two pass Merkle-Damgård

construction where message blocks are first processed in natural order and then, additionally, in reversed order. For such multi-round iterative hash functions we extend our model of execution graphs to include special round-nodes that partition the computation into multiple rounds. In each round, each message block m_i must be processed by an h-node. Furthermore, the output of round_i must be processed by an h-node in the next round $i+1$.

round-node There exist $r-1$ round-nodes with in-degree 1 taking an h-edge and out-degree 1. The outgoing edge is of type h-edge copying the label of the ingoing edge to the label of the single outbound edge. round-nodes partition the graph into distinct subgraphs and edges may not connect mp-nodes, hp-nodes, hmp-nodes or h-nodes in different subgraphs. We call the subgraph before the first round-node first round graph, the subgraph between the i-th and $i+1$-st round-node the $i+1$-st round graph and the subgraph after the $r-1$-st round-node the r-th round graph.

g-node The single g-node must be in the r-th round graph.

message-node For every message block m_i (for $1 \leq i \leq \ell$) there exists a node with in-degree 0 and out-degree at least r. For each message-node and round graph i there must be at least a single outbound m-edge connecting a mp node in the i-th round graph. Outgoing edges are of type m-edge and labeled with value m_i (possibly post- or prefixed with the message block counter or round counter) assuming the edge goes into the j-th round graph.

With this extended definition of execution graphs we can now define a notion of multi-round iterative hash functions.

Definition B.1. *Let the setup be as in the previous Definition 3.1. We call iterative hash function* $H^{\mathtt{h}}_{r,\mathtt{g},\mathtt{mp},\mathtt{hp},\mathtt{hmp},\mathtt{pad}} : \{0,1\}^* \to \{0,1\}^n$ *an r-round iterative hash function, if corresponding algorithm* construct *generates execution graphs containing $r-1$* round*-nodes.*

When functions g, mp, hp, hmp and pad are clear from context we simply write $H^{\mathtt{h}}_r$. Note that, this definition naturally extends the previous definition as the two are equivalent for $r = 1$.

B.4 Properties of Iterative Hash Functions

B.4.1 A Missing Link in $H^{\mathtt{h}}$

In the following we prove the missing link lemma from Section 3 which intuitively states, that if an adversary does not make all queries in the chain of a correct hash computation for some message M, then it has only negligible chance of learning the hash value of M.

Lemma 3.3 (restated). *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* h*. Let $\mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^\lambda; r))$ denote the adversary's queries to oracle* h *when algorithm $\mathcal{A}$ runs on randomness r and by $\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))$ denote the* h*-queries during the evaluation of $H^{\mathtt{h}}(M)$. Then it holds that*

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

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 h *and the coins of $\mathcal{A}$.*

Proof. Assume adversary $\mathcal{A}$ succeeds, that is, it outputs a message M and value y such that $H^{\mathtt{h}}(M) = y$ and there exists an h-query (m, x) which occurs during the evaluation of $H^{\mathtt{h}}(M)$ but which was not queried by $\mathcal{A}$. We consider the execution graph $\mathtt{pg} \leftarrow \mathtt{construct}(M)$ which induces a partial order on its nodes. That is, if n_1 and n_2 are nodes in pg then we write $\mathsf{n}_2 \succ \mathsf{n}_1$ if, and only if, there exists a directed path from node n_1 to node n_2 in pg. Relative to oracle h we can identify an h-node in graph pg for which the two input edges transport values m and x (that is, the edges will be labeled with values m and x when applying the generic algorithm $\mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$). We call this node $\mathtt{node}_{m,x}$ and in case there are multiple choices we simply choose one at random. As $\mathcal{A}$ does not query $\mathtt{h}(m, x)$ it holds that this value has min-entropy s-bits, that is, $\mathrm{H}_\infty(\mathtt{h}(m, x)) = s$. Value

$\mathtt{h}(m,x)$ is transported on all outgoing edges from $\mathtt{node}_{m,x}$. Each of these edges is connected to an $\mathtt{hp}$ or an $\mathtt{hmp}$-edge. By definition we have that

$$\mathrm{H}_\infty(\mathtt{hp}(U_s)) = \mathrm{H}_\infty(\mathtt{hmp}(U_s)) = s$$

where U_s is a random variable uniformly distributed in U_s. In other words, as preprocessing functions $\mathtt{hp}$ and $\mathtt{hmp}$ are injective they do not decrease entropy. Thus the sole outgoing edge of the preprocessing node transports s-bits of min-entropy to an $\mathtt{h}$-node $\mathtt{node}^*$ for which $\mathtt{node}^* \succ \mathtt{node}_{m,x}$.

Let $\mathtt{node}_{\mathtt{res}}$ denote the final $\mathtt{h}$-node in graph $\mathfrak{pg}$. As for any $\mathtt{h}$-node $\mathtt{node}'$ in $\mathfrak{pg}$ it holds

$$\mathtt{node}' = \mathtt{node}_{\mathtt{res}} \quad \vee \quad \mathtt{node}_{\mathtt{res}} \succ \mathtt{node}'$$

we get by recursively repeating the above argument that one of the input edges to $\mathtt{node}_{\mathtt{res}}$ transports s bits of min-entropy. Let $(m_{\mathtt{res}}, x_{\mathtt{res}})$ be the values transported on the two edges going into the final $\mathtt{h}$-node $\mathtt{node}_{\mathtt{res}}$. Then, we have that the probability that $\mathcal{A}$ queries $\mathtt{h}$ on $(m_{\mathtt{res}}, x_{\mathtt{res}})$ if it did not query $\mathtt{h}(m,x)$ is upper bounded by $q_{\mathcal{A}} \cdot 2^{-s}$.

By the above discussion we also directly yield that $\mathrm{H}_\infty(\mathtt{h}(m_{\mathtt{res}}, x_{\mathtt{res}})) = s$. Thus the probability of $\mathcal{A}$ guessing value y such that $\mathtt{g}(\mathtt{h}(m_{\mathtt{res}}, x_{\mathtt{res}})) = y$ is upper bounded by $2^{-\mathrm{H}_\infty(\mathtt{g}(U_s))}$ where again U_s denotes a random variable uniformly distributed in $\{0,1\}^s$. □

The last lemma can be strengthened by showing that if an adversary does not make query $(m,x) \in \mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))$ for some message M then it will not be able to make any query (m',x') for which $\mathtt{node}_{m',x'} \succ \mathtt{node}_{m,x}$. Here we denote by $\mathtt{node}_{m,x}$ the set of $\mathtt{h}$-nodes in the execution graph $\mathtt{construct}(M)$, for which the two input edges are labeled with values m and x relative to function $\mathtt{h}$. By $\succ$ we denote a partial order function and write

$$\mathtt{node}_{m',x'} \succ \mathtt{node}_{m,x}$$

if there exists $\mathtt{n}' \in \mathtt{node}_{m',x'}$ and $\mathtt{n} \in \mathtt{node}_{m,x}$ such that there is a directed path from n' to n in graph $\mathtt{construct}(\mathrm{M})$. Note that this is all relative to a function $\mathtt{h}$.

Lemma B.2. *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 the adversary's queries to oracle $\mathtt{h}$ when algorithm $\mathcal{A}$ runs on randomness r and $\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))$ the $\mathtt{h}$-queries during the evaluation of $H^{\mathtt{h}}(M)$. Then it holds that*

$$\mathrm{Pr}_{r,\mathtt{h}}\left[\begin{array}{l} (M,m,x) \leftarrow \mathcal{A}^{\mathtt{h}}(1^\lambda; r) \ : \\ \end{array} \begin{array}{r} (m,x) \in \mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M)) \wedge \\ \exists (m',x') \in \mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M)) : \Big((m',x') \notin \mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^\lambda; r)) \wedge \\ \mathtt{node}_{m,x} \succ \mathtt{node}_{m',x'} \Big) \end{array}\right] \leq \frac{q_{\mathcal{A}}}{2^s}$$

The probability is over the choice of random oracle $\mathtt{h}$ and the coins r of $\mathcal{A}$.

Proof. The proof follows with the same argument as in the proof of Lemma 3.3. Let for a message M and graph $\mathtt{construct}(M)$ and fixed-length random oracle $\mathtt{h}$

$$\mathtt{node}_{m,x} \succ \mathtt{node}_{m',x'}$$

that is, there exists a path from a node $\mathtt{n}' \in \mathtt{node}_{m',x'}$ to a node $\mathtt{n} \in \mathtt{node}_{m,x}$ in graph $\mathtt{construct}(M)$ relative to function $\mathtt{h}$ (note that we defined $\mathtt{node}_{m,x}$ as sets). If (m',x') is not queried by $\mathcal{A}$ to $\mathtt{h}$, that is $(m',x') \notin \mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{\mathtt{h}}(1^\lambda; r))$ then we have that value $\mathtt{h}(m',x')$ is a random variable with s bits of min-entropy. Furthermore, since this value is "passed down the graph" we have, by a recursive argument, that the input-edge-labels (relative to $\mathtt{h}$) for any $\mathtt{h}$-node $\mathtt{n}$ for which $\mathtt{n} \succ \mathtt{n}'$ is again a random variable with at least s bits min-entropy. Thus, the probability of an adversary $\mathcal{A}$ querying $\mathtt{h}$ on (m,x) such that (m,x) are the labels on the input edges of node $\mathtt{n}$ (relative to $\mathtt{h}$) is upper bounded by $q_{\mathcal{A}} \cdot 2^{-s}$. □

Next, we give a second version of the missing link lemma where we now let the adversary also gets access to the actual hash construction $H^{\mathtt{h}}$. In this case we can give a reduction to an indifferentiability simulator. In other words, if the hash construction is indifferentiable from a random oracle, no such adversary exists.

Lemma B.3. *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 $\mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}$ be an adversary that makes at most $q_{\mathcal{A}}$ many queries to $\mathtt{h}$ and $H^{\mathtt{h}}$, respectively. Let $\mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda;r))$ denote the adversary's queries to oracle $\mathtt{h}$ when algorithm $\mathcal{A}$ runs on randomness r, let $\mathsf{qry}^{H^{\mathtt{h}}}(\mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda;r))$ denote the adversary's queries to $H^{\mathtt{h}}$ and let $\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M))$ denote the $\mathtt{h}$-queries during the evaluation of $H^{\mathtt{h}}(M)$. Then for any indifferentiability simulator $\mathcal{S}$ there exists a distinguisher $\mathcal{D}$ such that*

$$\Pr_{r,\mathtt{h}}\left[(M,y) \leftarrow \mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda;r) : \begin{array}{c} H^{\mathtt{h}}(M) = y \wedge M \notin \mathsf{qry}^{H^{\mathtt{h}}}(\mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda;r)) \wedge \\ \left(\mathsf{qry}^{\mathtt{h}}(H^{\mathtt{h}}(M)) \setminus \mathsf{qry}^{\mathtt{h}}(\mathcal{A}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda;r))\right) \neq \emptyset \end{array}\right] \leq \mathsf{Adv}^{\mathit{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}) + \frac{4q_{\mathcal{A}}^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))}}$$

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 $\mathcal{A}$.

Proof. Assume the adversary succeeds and let (M,y) be its output. Then the result query (m,x) for the corresponding execution graph $\mathtt{construct}(M)$ must have been asked by $\mathcal{A}$ to $\mathtt{h}$ but for a guessing probability of $2^{-\mathrm{H}_\infty(\mathtt{g}(U_s))}$. For this note that $\mathtt{h}$ is ideal and $\mathtt{g}(\mathtt{h}(m,x))$ has min-entropy $\mathrm{H}_\infty(\mathtt{g}(U_s))$, where U_s denotes a random variable uniformly distributed in $\{0,1\}^s$. This, however, means that the result of at least one of the left out queries can be reconstructed from the queries of $\mathcal{A}$ to $\mathtt{h}$. Let us by $Q_{\mathcal{A}}$ denote the queries of $\mathcal{A}$ to $\mathtt{h}$. Then the result of the missing query must be $\mathtt{hp}^{-1}(x)$ or $\mathtt{hmp}^{-1}(m)$ for some $(m,x) \in Q_{\mathcal{A}}$. As $\mathtt{h}$ is ideal, the probability of guessing this value is 2^{-s}.

From an $H^{\mathtt{h}}(M')$ for some message M' the adversary learns the outcome of the corresponding result query (m,x), that is it learns $\mathtt{g}(\mathtt{h}(m,x))$. If this value corresponds to the result of the missing query in M, then we have that, unless the adversary found a collision on $\mathtt{h}$ (which happens at most with probability $q_{\mathcal{A}}^2 \cdot 2^{-s+1}$) that the execution graph $\mathtt{pg}' \leftarrow \mathtt{construct}(M')$ (without the final $\mathtt{g}$-node) is a subgraph of the execution graph of $\mathtt{pg} \leftarrow \mathtt{construct}(M)$. Note that this is a property of the hash construction H and independent of the ideal function $\mathtt{h}$ that it is used with. That is, for $H^{\mathtt{h}}$ the execution graph of M' is always a subgraph of the execution graph of M. This we will later exploit to construct an indifferentiability distinguisher. Furthermore, note that if the adversary is able to recover the result of the result query from $\mathtt{g}(\mathtt{h}(m,x))$ this means that the preimage space $\mathtt{g}^{-1}(\mathtt{g}(\mathtt{h}(m,x)))$ must be at most polynomial in size since any preimage has the same probability of being the *correct* one. Note that this is the reason why, for example, chop-MD is indifferentiable. Here function $\mathtt{g}$ only outputs the first half of the bits and thus the preimage space $\mathtt{g}^{-1}(\mathtt{g}(\mathtt{h}(m,x)))$ is exponential.

We now build a distinguisher $\mathcal{D}$ in the indifferentiability game. Distinguisher $\mathcal{D}$ gets access to functionality $\mathcal{F} := (\mathcal{F}.\mathsf{hon}, \mathcal{F}.\mathsf{adv})$ which is either $(H^{\mathtt{h}},\mathtt{h})$ in the real world, or $(\mathcal{R},\mathcal{S}^{\mathcal{R}})$ in the ideal world and where $\mathcal{S}$ is some indifferentiability simulator and $\mathcal{R}$ is a random oracle. Distinguisher $\mathcal{D}$ lazily samples an ideal function $\mathtt{h}'$. It runs adversary $\mathcal{A}$ giving it access to $H^{\mathtt{h}'}$ and $\mathtt{h}'$. Note that this computation is completely local to $\mathcal{D}$. If the adversary succeeds, $\mathcal{D}$ extracts messages M and M' such that the execution graph of M is a subgraph of M'. It constructs the corresponding execution graphs $\mathtt{pg} \leftarrow \mathtt{construct}(M)$ and $\mathtt{pg}' \leftarrow \mathtt{construct}(M')$. It then consistently relabels all message nodes with random values (such that $\mathtt{pg}$ remains a subgraph of $\mathtt{pg}'$) and extracts the resulting messages $M \leftarrow \mathtt{extract}(\mathtt{pg})$ and $M' \leftarrow \mathtt{extract}(\mathtt{pg}')$.

Distinguisher $\mathcal{D}$ then queries its left oracle $\mathcal{F}.\mathsf{hon}$ (which is either $H^{\mathtt{h}}$ or the random oracle $\mathcal{R}$) on message M to receive y and on message M' to receive y'. Finally, for all $x \in \mathtt{g}^{-1}(y)$ it computes the remainder of the execution graph for M' using its right oracle (which is either $\mathtt{h}$ or $\mathcal{S}^{\mathcal{R}}$) starting from the final node of the subgraph for M' and using value x as the label of the outgoing edge of that node. If for any $x \in \mathtt{g}^{-1}(y)$ the two computations match it outputs 1, else it outputs 0.

ANALYSIS. Assuming that the adversary is successful and did not guess value y or find an $\mathtt{h}$ collision the distinguisher will succeed in extracting the two messages as described. In the real world, the computation will always match and the distinguisher will, thus, output 1 with probability 1. In the ideal world, however, as the simulator will not see the queries related to message M, its probability of correctly guessing value $\mathcal{R}(M')$ is at most $2^{-|M|}$. As we can assume that M is at least one message block long we have that this probability is less than 2^{-d}. Note that we can further optimize this probability (for example to 2^{-s}) by repeating the checking steps, that is, repeatedly relabeling the graphs' nodes and repeat the checking. Thus, we can estimate

the success probability of adversary $\mathcal{A}$ as

$$\Pr[\,\mathcal{A} \text{ is successful}\,] \leq \mathsf{Adv}^{\text{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}) + \frac{q_{\mathcal{A}}^2}{2^{s-1}} + \frac{1}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))}} + \frac{1}{2^s} \leq \frac{4q_{\mathcal{A}}^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))}}$$

□

B.4.2 Extractor for Hash Function $H^{\mathtt{h}}$

We now prove the extractor lemma from Section 3. Note that the extractor is such, that it reconstructs exactly those messages M for which an adversary "knows" the corresponding hash value.

Lemma 3.4 (restated). *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*

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

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 $\mathtt{h}$.

Proof. We will first present extractor $\mathcal{E}$ (see Figure 12) to then argue that it achieves the claimed bound. Extractor $\mathcal{E}$ will work with partial graphs that we will store in a set $\mathcal{PG}$. Without loss of generalization we assume that Q does not contain the same query twice, that is $\mathcal{A}$ does repeat queries to $\mathtt{h}$.

The extractor will do a single pass over query sequence Q. In each step extractor $\mathcal{E}$ initializes new partial graphs if the current query is an initial query. A freshly initialized graph consists of either an $\mathcal{IV}$-node connected to a $\mathtt{hp}$-node, a message-block-node (or possibly a second $\mathcal{IV}$-node) connected to a $\mathtt{mp}$-node and an $\mathtt{h}$-node which is connected to the $\mathtt{mp}$ and $\mathtt{hp}$ nodes. The outgoing $\mathtt{h}$-edge is free. We denote the creation of new partial graphs for query (m, x) by

$$\textbf{new}\ \text{PartialGraph}(\mathtt{mp}^{-1}(m), \mathtt{hp}^{-1}(x))$$

Whenever a new partial graph is constructed (or later extended) we compute the value of the sole outgoing $\mathtt{h}$-edge and denote it by $\mathtt{pg}.\mathsf{y} \leftarrow y$. Note that, by construction we have that $\mathtt{pg}.\mathsf{y} = y = \mathtt{EVAL}^{\mathtt{h}}(\mathtt{pg})$.

For each query $(m, x, y) \in Q$ (where (m, x) denotes the query and y the oracle answer) additionally to checking if it is an initial query we try to extend existing partial graphs in $\mathcal{PG}$ by this query. A partial graph $\mathtt{pg} \in \mathcal{PG}$ can be extended if its sole outgoing $\mathtt{h}$-edge which transports value $\mathtt{pg}.\mathsf{y}$ can be connected to a new $\mathtt{h}$-node constructed from query (m, x). This is the case, if and only if, (i) $\mathtt{pg}.\mathsf{y} = \mathtt{hp}^{-1}(x)$ or if (ii) $\mathtt{pg}.\mathsf{y} = \mathtt{hmp}^{-1}(m)$. Note that for the second case, where $\mathtt{pg}.\mathsf{y} = \mathtt{hmp}^{-1}(m)$ we can only extend the partial graph if there is also $\mathtt{pg}' \in \mathcal{PG}$ such that $\mathtt{pg}'.\mathsf{y} = \mathtt{hp}^{-1}(x)$ as otherwise the $\mathtt{hp}$-node would not have all its input edges bound, thus the new partial graph is constructed from two previously existing graphs.[5] If we extend a partial graph, a new partial graph is generated for the extended graph and the old one is kept in $\mathcal{PG}$. We denote by

$$\mathtt{pg}.\text{extendedBy}(m, x)$$

the partial graph generated from $\mathtt{pg}$ and extended by query $(m, x, y) \in Q$ (corresponding to case (i), see above) and by

$$\mathtt{pg}.\text{extendedBy}(\mathtt{pg}', m, x)$$

[5] See the HMAC or NMAC construction (Appendix B.5) for an example, where this case can occur.

Extractor: $\mathcal{E}(Q)$

$\mathcal{M} \leftarrow \{\}; \mathcal{Y} \leftarrow \{\}; \mathcal{PG} \leftarrow \{\}$
for $i = 1 \ldots |Q|$ **do** /* building partial graphs */
  $(m, x, y) \leftarrow Q[i]$
  if $\mathsf{init}(m, x)$ **then**
    $\mathfrak{newG} \leftarrow$ **new** PartialGraph$(\mathtt{mp}^{-1}(m), \mathtt{hp}^{-1}(x))$
    $\mathfrak{newG}.\mathsf{y} \leftarrow y$
    $\mathcal{PG} \leftarrow \mathcal{PG} \cup \mathfrak{newG}$
  /* try to extend partial graphs */
  foreach $\mathfrak{pg} \in \mathcal{PG} : \mathfrak{pg}.\mathsf{y} = \mathtt{hp}^{-1}(x)$ **do**
    $\mathfrak{newG} \leftarrow \mathfrak{pg}.\mathrm{extendedBy}(m, x)$
    $\mathfrak{newG}.\mathsf{y} \leftarrow y$
    $\mathcal{PG} \leftarrow \mathcal{PG} \cup \mathfrak{newG}$
  foreach $(\mathfrak{pg}, \mathfrak{pg}') \in \mathcal{PG} \times \mathcal{PG} : \mathfrak{pg}.\mathsf{y} = \mathtt{hmp}^{-1}(m) \wedge \mathfrak{pg}'.\mathsf{y} = \mathtt{hp}^{-1}(x)$ **do**
    $\mathfrak{newG} \leftarrow \mathfrak{pg}.\mathrm{extendedBy}(\mathfrak{pg}', m, x)$
    $\mathfrak{newG}.\mathsf{y} \leftarrow y$
    $\mathcal{PG} \leftarrow \mathcal{PG} \cup \mathfrak{newG}$
foreach $\mathfrak{pg} \in \mathcal{PG}$ **do** /* building target message set */
  $M \leftarrow \mathtt{extract}(\mathfrak{pg})$
  if $M \neq \perp$ **do**
    $\mathcal{M} \leftarrow \mathcal{M} \cup M$
    $\mathcal{Y} \leftarrow \mathcal{Y} \cup g(\mathfrak{pg}.y)$
return $(\mathcal{M}, \mathcal{Y})$

Figure 12: The extractor for Lemma 3.4.

the partial graph generated from the two partial graphs $\mathfrak{pg}$ and $\mathfrak{pg}'$ extended by query $(m, x, y) \in Q$ which corresponds to case (ii). Again, after a new graph is constructed we set $\mathfrak{pg}.\mathsf{y} \leftarrow y$. Note that also here, by construction, we have that $\mathfrak{pg}.\mathsf{y} = y = \mathtt{EVAL}^{\mathtt{h}}(\mathfrak{pg})$.

After all partial graphs are constructed the extractor then recovers for each partial graph the sequence of message-blocks using algorithm `extract`. These, form the set of target messages output by extractor $\mathcal{E}$. Furthermore, if $\mathtt{extract}(\mathfrak{pg})$ outputs a message M, then, by construction we have that $g(\mathfrak{pg}.y) = \mathtt{EVAL}^{\mathtt{h}}(\mathfrak{pg}) = H^{\mathtt{h}}(M)$. This forms the set of target hash values $\mathcal{Y}$. We give the pseudo-code for extractor $\mathcal{E}$ in Figure 12.

It remains to argue that extractor $\mathcal{E}$ has the claimed runtime, as well as that the target set $\mathcal{M}$ as output by $\mathcal{E}$ is sufficient. For the run-time note that the extractor makes a single pass over the query set Q. If adversary $\mathcal{A}$ did not find collisions in $\mathtt{h}$ (which only happens with probability less than $q_{\mathcal{A}}^2 \cdot 2^{-s+1}$) in each step at most 3 new partial graphs are generated. Thus, with overwhelming probability the number of generated partial graphs is at most $3|Q| = 3q_{\mathcal{A}}$ (for the case that more than $3q_{\mathcal{A}}$ partial graphs are found we assume that $\mathcal{E}$ stops and the adversary wins which corresponds to parts of the first term in the statement of Lemma 3.4). For each of the partial graphs we run `extract` once, which leaves us with a runtime of $\mathcal{O}(3q_{\mathcal{A}} \cdot t_e)$ where t_e denotes the run-time of deterministic algorithm `extract`.

Let us now show that $\mathcal{M}$ is sufficient. By Lemma 3.3 we have that all $\mathtt{h}$-queries occurring during the computation of $H^{\mathtt{h}}(M)$ must be in Q but for probability

$$\frac{q_{\mathcal{A}}}{2^s} + \frac{1}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))}}$$

Furthermore, the queries appear in the correct order but for probability $\frac{q_{\mathcal{A}}}{2^s}$ (see Lemma B.2).

Putting it all together, we have that if $\mathcal{A}$ does not find any collision on $\mathtt{h}$ (which occurs at most with probability $q_{\mathcal{A}}^2 \cdot 2^{-s+1}$) then for any M output by $\mathcal{A}$ all $\mathtt{h}$-queries in $H^{\mathtt{h}}(M)$ must be in Q but for the guessing probability

$$\frac{q_{\mathcal{A}}}{2^s} + \frac{1}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))}} \ .$$

Furthermore, the queries must appear in topologically correct order but for probability $q_{\mathcal{A}} \cdot 2^{-s}$. Then, however, the corresponding message M is reconstructed also by $\mathcal{E}$ as by definition it reconstructs partial graphs if all queries appear in topologically correct order. □

B.5 Examples: Hash Constructions in Compliance with Definition 3.1

B.5.1 Merkle-Damgård-like Functions

In the following we show that Merkle-Damgård-like functions such as the plain or chop-MD constructions, are covered by Definition 3.1. The difference between chop-MD and the plain Merkle-Damgård construction only lies in the final transformation g which is the identity for plain Merkle-Damgård and which truncates the output of the final compression function call in case of chop-MD.

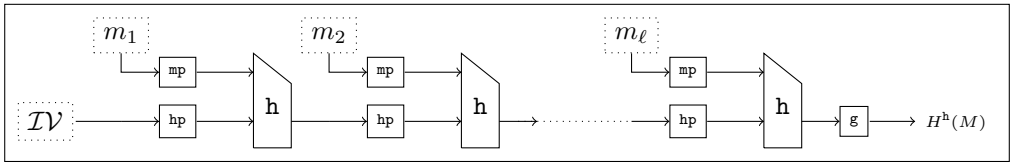

Figure 13: Merkle-Damgård Construction

Merkle-Damgård constructions use a single $\mathcal{IV}$ that is connected to the first hp-node. Given message blocks $m_1 \| \ldots \| m_\ell = \mathtt{pad}(M)$ for $M \in \{0,1\}^*$ it is easy to see that algorithm `construct`can construct the corresponding execution graph depicted in Figure 13 in time linear in the number of message blocks, that is $\mathcal{O}\left(\frac{|M|}{d}\right)$. Similarly checking an execution graph for validity and extracting the message from a valid (partial) graph can be done in time linear in the number of nodes.

B.5.2 NMAC and HMAC

In the following we show how NMAC and HMAC [BCK96] fit into Definition 3.1. NMAC and HMAC are originally keyed hash constructions, but can be "de-keyed" by fixing the key to a constant initialization vector. We here give the formalization NMAC in the unkeyed setting and for HMAC in the keyed setting. Note that the only difference is that for the keyed setting we exchange the corresponding $\mathcal{IV}$ nodes for κ-nodes (see Section B.2).

HMAC and NMAC are the first constructions where we use the hmp preprocessing function. The running times of `construct` and `extract` are equivalent to the running times for the basic Merkle-Damgård constructions.

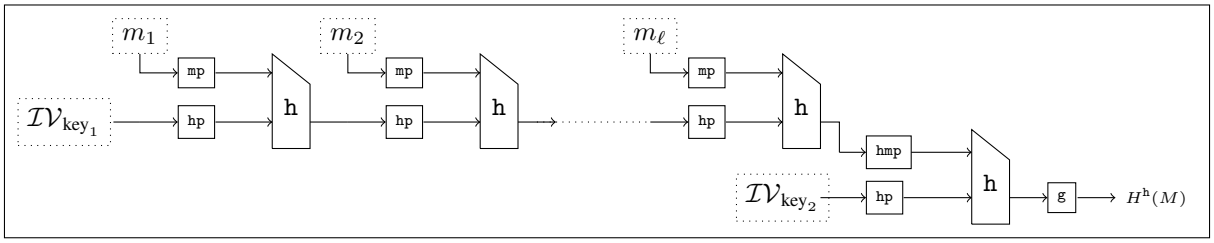

Figure 14: NMAC: note the use of the hmp node to connect the final h-node to the plain Merkle-Damgård construction.

B.5.3 Hash Tree

In Figure 16 we show how hash trees fit into Definition 3.1. For simplicity we assume that the number of message blocks ℓ is a power of 2. Given message blocks $m_1 \| \ldots \| m_\ell = \mathtt{pad}(M)$ for $M \in \{0,1\}^*$ it is easy to see that we can construct the corresponding execution graph in time log-linear in the number of message blocks, that is $\mathcal{O}(\ell \log \ell)$. Extracting the message of a given partial graph can be done in time of a (reverse) breadth first search starting from the final h-node.

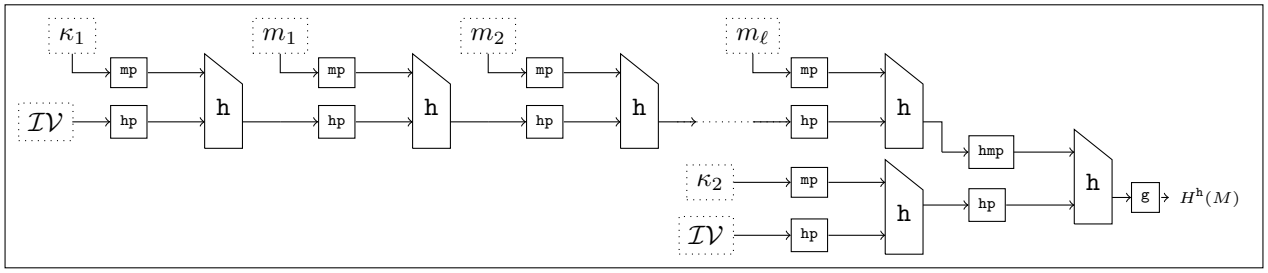

Figure 15: The HMAC construction in the keyed setting with κ-nodes κ_1 and κ_2 (see Section B.2). Note that HMAC is similar to NMAC except that the actual keys are now generated by h-calls.

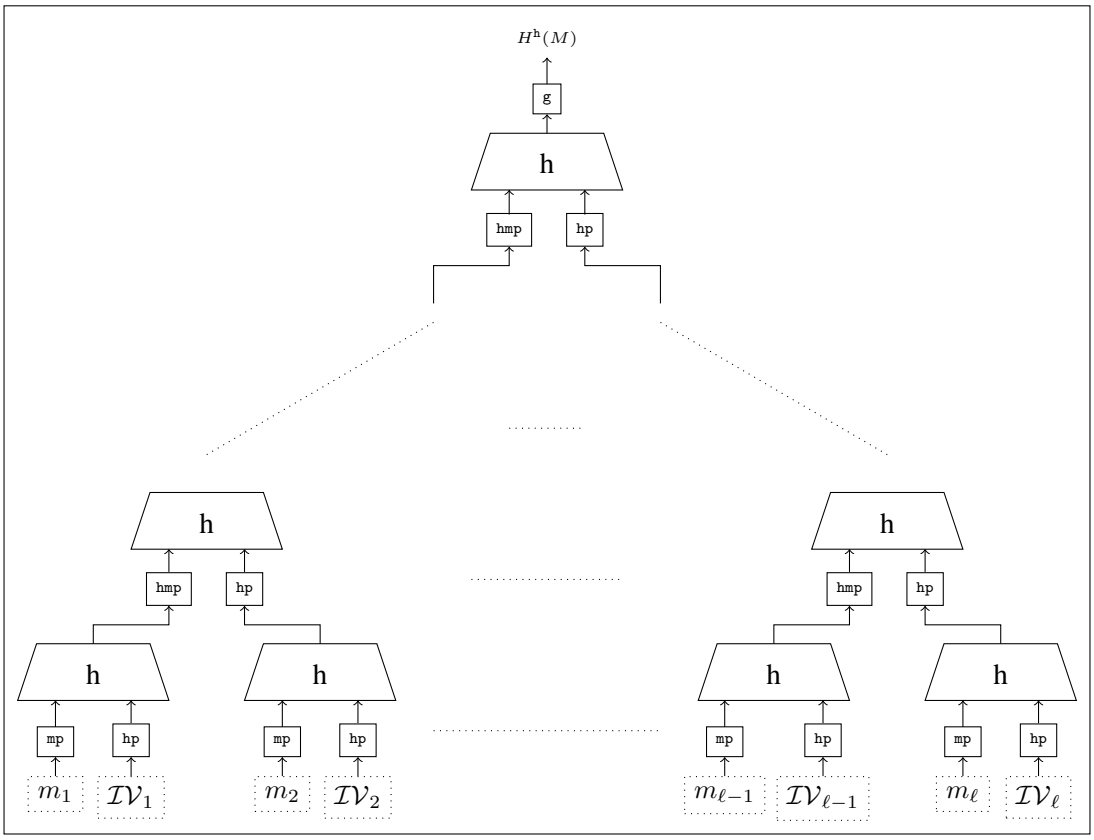

Figure 16: A hash tree in the formalization of Definition 3.1.

B.5.4 The Double-Pipe Construction / Extensions to the Model

Stefan Lucks [Luc04] proposes several tweaks to the design of iterated hash functions to, for example, rule out generic attacks such as Joux' multi-collision attack [Jou04]. In Figure 17 we show how the double-pipe construction fits into Definition 3.1. Note that to support constructions like the double-pipe construction we must slightly extend our model of iterative hash functions. We must now allow that hp-nodes not only have in-degree 1 but 2. Besides slightly complicating the definition of initial queries and chained queries —we now have to split the pre-image of $\mathtt{hp}^{-1}(x)$ into multiple values— the proofs and intuition presented in this paper work analogously.

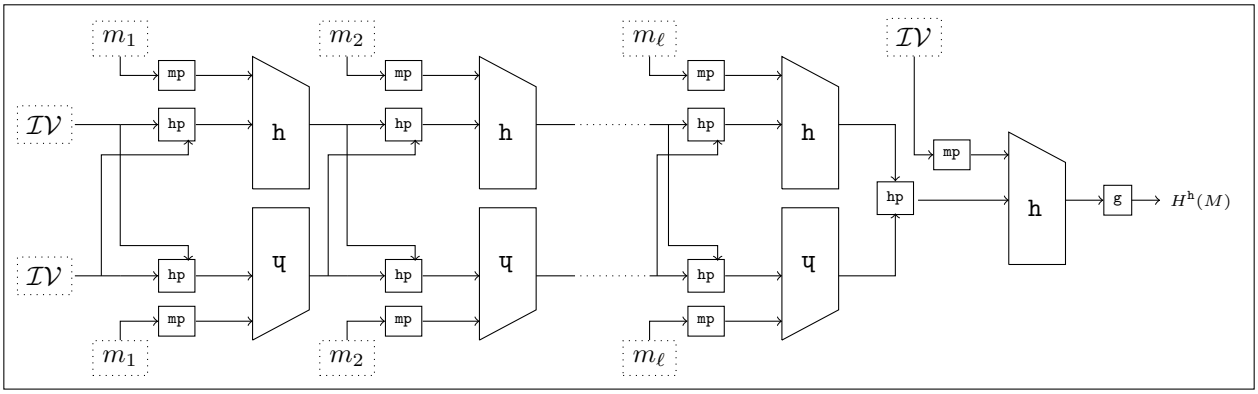

Figure 17: The double-pipe construction from [Luc04]. Note that the message-block-nodes are split for better drawing. They should, however, be regarded as a single node.

B.6 The Sponge Construction

In the following section we describe what changes need to be done to the model to capture the sponge construction [BDPA11b] used in SHA-3. The sponge construction is one of the few exceptions that does not iterate a compression function, but a permutation $\mathtt{f} : \{0,1\}^n \to \{0,1\}^n$. To process a message $m_1 \| \ldots \| m_\ell = \mathtt{pad}(M)$ sponge starts with an empty state $\mathcal{IV} = 0^n$ which is split into the bitrate r and capacity c Messages are shorter than the state—in SHA3 one of the recommended ratios is 576 bit messages and 1024 bit capacity resulting in a 1600 bit state—and in each round the current message block is xored onto the first part of the state r. The capacity is never touched by message blocks directly. After the message is xored onto the state the current state is run through the permutation $(r' \| c') \leftarrow f(r \| c)$ to complete the round. We depict the computation of sponge in Figure 18. Note that for Keccak the preprocessing functions hp and mp are the identity. The final transformation g outputs the first ℓ bits of the current state, for example 512 bits.[6]

Next we describe how to adapt the definition of initial, chained and result queries for sponge. To be consistent with our established notation we think of f as having two inputs $\mathtt{f} : \{0,1\}^r \times \{0,1\}^c \to \{0,1\}^r \times \{0,1\}^c$ where the first input corresponds to the first r bits of the state (the bitrate) and the second input to the remaining c bits (the capacity). A query (m, x) to f is an initial query if $\mathtt{hp}^{-1}(x) = \mathcal{IV}_c$, where $\mathcal{IV}_c$ denotes the capacity parts of the initialization vector. For chained and result queries the definitions remain the same.

CONSEQUENCES ON RESULTS. In all our proofs we use only three properties of ideal compression functions: 1) over the choice of compression function h the random variable $\mathtt{h}(m, x)$, for fixed m and x has high min-entropy. This is used to argue that hash values cannot be learned without querying all h-queries that also occur within an honest execution (see the missing link lemma; Lemma 3.3). 2) Function h is perfectly collision resistant, that is, finding collisions on h requires approximately $2^{s/2}$ many invocations where $\{0,1\}^s$ denotes image-space of h. This is used to show that an extractor exists that on seeing all h-queries by an adversary can reconstruct any execution graph, which also the adversary has access to (see the extractor lemma; Lemma 3.4 as well as Lemma C.2). Finally, 3) we require that a hash value $H^{\mathtt{h}}(M)$ for a message M does not leak any more information on queries to h during the computation of $H^{\mathtt{h}}(M)$ than does message M (see, for example, Lemma B.3).

We expect our results also to hold for the sponge setting where the ideal compression function is exchanged for an ideal permutation (and the adversary might also have access to its inverse) and in the following paragraph present our reasoning. We note, however, that we have not formally verified this claim.

Clearly, an ideal permutation fulfills the high-entropy requirement. For the collision resistance note, that in the sponge construction we are concerned with collisions on the capacity part (that is the last c output bits). Thus, similarly to compression functions, an adversary must make approximately $2^{c/2}$ queries to find a collision. Note, however, that the adversary also has access to the inverse of permutation f and can use it to generate collisions. For this note that the input f within an execution graph is only partly controlled by

[6] If one combines the xoring of the message block with the execution of the permutation, then the sponge construction is in fact a chopMD construction with compression function $\mathtt{h}(m, (r|c)) := f(r \oplus m|c)$.

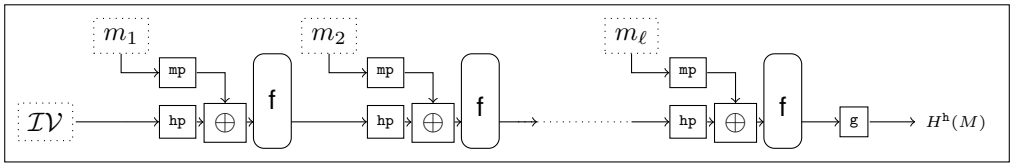

Figure 18: The Sponge Construction. A state split into rate and capacity is iteratively processed through a permutation f. In each round the current message block is xored onto the first r bits of the state. The remaining c bits (the sponge's capacity) is never directly influenced by a message block. The final transformation g outputs a substring of the first r bits of the state.

the adversary, that is, the capacity is never touched directly. Thus, in order to "connect" to partial execution graphs via an inverse query to f requires the adversary to find a target collision for which we can directly upper bound the probability by $q2^{-s}$ where q denotes the number of queries by the adversary. Finally, for the third requirement note that a hash value $H^{\mathtt{h}}(M)$ is computed as $\mathtt{g}(\mathtt{h}(m,x))$ for some result query (m,x). In the case of sponge g is greatly compressing— in SHA-3, for example, from 1600 bits to 512—and, hence value $\mathtt{h}(m,x)$ cannot be reconstructed given only $\mathtt{g}(\mathtt{h}(m,x))$ even with unbounded resources.

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 $\quad \mathcal{T} \leftarrow \mathcal{T} \cup$ **new** PartialGraph$(\mathtt{mp}^{-1}(m), \mathtt{hp}^{-1}(x))$
5 /* extend existing graphs */
6 **foreach** $(\mathtt{pg}, \mathtt{pg}') \in \mathcal{PG} \times \mathcal{PG} : \mathtt{pg.y} = \mathtt{hmp}^{-1}(m) \wedge \mathtt{pg}'.\mathtt{y} = \mathtt{hp}^{-1}(x)$ **do**
7 $\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathtt{pg}.\mathrm{extendedBy}(\mathtt{pg}', m, x)$
8 **foreach** $\mathtt{pg} \in \mathcal{PG} : \mathtt{pg.y} = \mathtt{hp}^{-1}(x)$ **do**
9 $\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathtt{pg}.\mathrm{extendedBy}(m, x)$
10 /* test for conditions B_2, B_3 and if true then abort */
11 **if** $\exists \mathtt{pg} \in \mathcal{T} : \mathtt{extract}(\mathtt{pg}) \neq \perp$ **then**
12 $\quad \mathcal{M}[m, x] \leftarrow_{\$} \mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg})))$
13 **else**
14 $\quad \mathcal{M}[m, x] \leftarrow \{0, 1\}^s$
15 /* test for condition B_1 and if true then abort */
16 **foreach** $\mathtt{pg} \in \mathcal{T}$ **then**
17 $\quad \mathtt{pg.y} \leftarrow \mathcal{M}[m, x]$
18 $\quad \mathcal{PG} \leftarrow \mathcal{PG} \cup \mathtt{pg}$
19 **return** $\mathcal{M}[m, x]$;

Figure 19: Simulator $\mathcal{S}_*$ from Construction C.1 as pseudo-code. The conditions named in comments in lines 10 and 15 are defined in the second game hop in the proof of Lemma C.2.

C The Composition Theorem 4.2

In this section we present the proof of Theorem 4.2. The proof appearing on page 42 makes use of a generic simulator and a derandomization technique for this simulator. We present the generic simulator in the upcoming Section C.1 and the derandomization step in Section C.2. Once having established these results we present the complete proof of Theorem 4.2 in Section C.3. A proof sketch and the outline of these three steps is given in Section 4.1.

C.1 A Generic Indifferentiabilitiy Simulator

In the following we show that for any indifferentiable hash construction we can use a generic simulator which replies with randomly chosen values on any non result query and only on result queries picks a value consistent with the random oracle. For this we consider the following simulator $\mathcal{S}_*$ that uses similar techniques to that of the extractor of Lemma 3.4 (see Figure 12). We also use a similar syntax as for the proof of Lemma 3.4.

Construction C.1. *We give the pseudo-code of simulator $\mathcal{S}_*$ when receiving query (m, x) in Figure 19. We assume that it initializes table $\mathcal{M} \leftarrow []$ and set $\mathcal{PG} \leftarrow \{\}$ before processing the first query.*

As in Lemma 3.4 we denote the value of the sole outgoing h-edge of a partial graph $\mathtt{pg}$, relative to an execution with simulator $\mathcal{S}_*$, by $\mathtt{pg.y}$ (which is assigned in line 17). This value is used, as in extractor $\mathcal{E}$ (see Lemma 3.4) to check whether partial graphs can be extended (lines 6 and 8).

A DESCRIPTION OF $\mathcal{S}_*$. We give the pseudo-code of simulator $\mathcal{S}_*$ in Figure 19. The Simulator is very similar to the extractor from Lemma 3.4. Simulator $\mathcal{S}_*$ keeps a table $\mathcal{M}$ for storing all the queries it received and a set $\mathcal{PG}$ for storing partial graphs that it creates. On receiving a query (m, x) simulator $\mathcal{S}_*$ checks table $\mathcal{M}$ whether the query has been queried before. If so it returns the same result as before: $\mathcal{M}[m, x]$. If it is a fresh query it takes similar steps as extractor $\mathcal{E}$ in Lemma 3.4 to generate new partial graphs which it temporarily stores in set $\mathcal{T}$. For this it checks whether the query is an initial query, or whether it extends any of the partial graphs that were generated during previous calls. To compute the output, the simulator checks if any of the generated partial graphs correspond to a valid execution graph, that is, whether the query was a result query

(see Definition 3.2 and description of model in Section 3). If this is the case, it picks a value uniformly at random from the preimage of $\mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathfrak{pg}))$ and sets

$$\mathcal{M}[m,x] \leftarrow\!\!\$\ \mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathfrak{pg}))$$

(see line 12). Again note that this execution branch corresponds to result queries and the simulator thus picks the value such that it is consistent with the random oracle. If, on the other hand, no partial graph was generated or no partial graph can be completed to a valid execution graph, that is, $\mathtt{extract}(\mathfrak{pg}) = \perp$ for all $\mathfrak{pg} \in \mathcal{T}$, then simulator $\mathcal{S}_*$ chooses a value $\mathcal{M}[m,x] \leftarrow \{0,1\}^s$ uniformly at random to answer the query (see line 14). Let us stress, that this value is generated independently of the state of the simulator, but it depends only on the query and the simulators random coins. Finally, for all generated partial graphs $\mathfrak{pg} \in \mathcal{T}$ the simulator sets value $\mathfrak{pg}.\mathrm{y}$ to $\mathcal{M}[m,x]$ and adds the newly created partial graph to its set of graphs $\mathcal{PG}$ (lines 17 and 18). It returns value $\mathcal{M}[m,x]$.

SIMULATOR $\mathcal{S}_*$ IS A GOOD INDIFFERENTIABILITY SIMULATOR. In the following we show, that simulator $\mathcal{S}_*$ as described above is a good indifferentiability simulator game as the underlying simulator $\mathcal{S}$ that it is build from. This is captured by the following lemma:

Lemma C.2. *Let $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ be an iterative hash function and $\mathcal{R}$ a random oracle. Let simulator $\mathcal{S}_*$ be constructed as in Construction C.1 that exports the same interface as $\mathtt{h} : \{0,1\}^k \times \{0,1\}^d \to \{0,1\}^s$. Then for any distinguisher $\mathcal{D}$ making at most q oracle queries there exists distinguisher $\mathcal{D}'$ such that*

$$\mathsf{Adv}^{\textit{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}_*}(\mathcal{D}) \le \mathsf{Adv}^{\textit{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}') + \frac{11q^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))-1}} \ .$$

Proof. We want to upper bound the indifferentiability advantage of a distinguisher $\mathcal{D}$ for simulator $\mathcal{S}_*$:

$$\mathsf{Adv}^{\mathrm{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}_*}(\mathcal{D}) = \left|\Pr\left[\mathcal{D}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda)=1\right] - \Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}_*^{\mathcal{R}}}(1^\lambda)=1\right]\right| \ .$$

For the proof we use a game based approach (similar to the indifferentiability proofs in [CDMP05]) starting from experiment $\Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}_*^{\mathcal{R}}}(1^\lambda)=1\right]$ in GAME_1 until we reach the target experiment $\Pr\left[\mathcal{D}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda)=1\right]$ in GAME_4 summing up the distinguishing probabilities in the individual game hops.

GAME_1. We start with the original security game:

$$\Pr[\,\mathrm{GAME}_1\,] = \Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}_*^{\mathcal{R}}}(1^\lambda)=1\right]$$

GAME_2. This game is as the previous game, but for a slightly changed simulator. The new simulator $\mathcal{S}_0$ works as simulator $\mathcal{S}_*$ but looks for conditions that might be exploited by a distinguisher and deliberately fails in such a situation. That is, $\mathcal{S}_0$ fails if one of the following failure conditions occur on receiving query (m,x):

Condition B_1: Simulator $\mathcal{S}_*$ generates an output value $\mathcal{M}[m,x]$ such that there exists $(m',x') \neq (m,x)$ for which $\mathsf{g}(\mathcal{M}[m',x']) = \mathsf{g}(\mathcal{M}[m,x])$.

Condition B_2: Simulator $\mathcal{S}_*$ generates two partial graphs $\mathfrak{pg}$ and $\mathfrak{pg}'$ that can both be completed, that is, $\mathtt{extract}(\mathfrak{pg}) \neq \perp$ and $\mathtt{extract}(\mathfrak{pg}') \neq \perp$.

Condition B_3: The simulator keeps an additional list $\mathcal{L}$ of all queries to it. When on a new query (m,x) a new partial graph $\mathfrak{pg}$ is generated it tests for all earlier queries $(m',x') \in \mathcal{L}$ whether $\mathfrak{pg}$ can be extended by (m',x'). If any query is found, the simulator fails.

Let GAME_2 be the event that distinguisher $\mathcal{D}$ outputs one in this setting, i.e.,

$$\Pr[\,\mathrm{GAME}_2\,] = \Pr\left[\mathcal{D}^{\mathcal{R},\mathcal{S}_0^{\mathcal{R}}}(1^\lambda)=1\right] \ .$$

The responses between the distinguisher in GAME_1 and GAME_2 can only differ, if the simulator reaches one of the failure conditions. For the difference between games GAME_1 and GAME_2 it holds that

$$|\Pr[\,\mathrm{GAME}_2\,] - \Pr[\,\mathrm{GAME}_1\,]| \leq \Pr\left[\bigcup_{i=1}^{3} B_i \text{ holds for any of the queries}\right]$$

We consider the failure conditions in turn.

For event B_1 note that the simulator chooses its outputs uniformly at random from $\{0,1\}^s$. Thus, we can directly estimate the probability of event B_1 as

$$\Pr[\,B_1 \text{ holds for any of the queries}\,] \leq \frac{q^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))-1}}$$

Failure condition B_2 corresponds to the simulator generating two partial graphs $\mathtt{pg}$ and $\mathtt{pg}'$ on one query that can both be completed to valid execution graphs. Note that, by definition, in this case $\mathtt{extract}(\mathtt{pg}) \neq \mathtt{extract}(\mathtt{pg}')$ and hence the simulator could not choose an output that is consistent with both partial graphs. This, however, directly implies a collision in the simulator's output and thus

$$\Pr[\,B_2 \text{ holds for any of the queries}\,] \leq \frac{q^2}{2^{s-1}}$$

For event B_3 note that for a partial graph $\mathtt{pg}$ to be extendable by an earlier query $(m', x') \in \mathcal{L}$ it must hold that $\mathtt{hp}^{-1}(x') = \mathtt{pg}.\mathrm{y}$.[7] As partial graph $\mathtt{pg}$ was generated only after query (m', x') was queried to the simulator by the distinguisher we can bound the probability of guessing value x' such that a later query to the underlying simulator $\mathcal{S}^{\mathcal{R}}$ yields value $\mathtt{hp}^{-1}(x')$ with the birthday bound as $q^2 \cdot 2^{-s+1}$. Thus,

$$\Pr[\,B_3 \text{ holds for any of the queries}\,] \leq \frac{q^2}{2^{s-1}}$$

Putting it all together (via a union bound) we have that

$$\begin{aligned} |\Pr[\,\mathrm{GAME}_2\,] - \Pr[\,\mathrm{GAME}_1\,]| &\leq \Pr\left[\bigcup_{i=1}^{3} B_i \text{ hold for any of the queries}\right] \\ &\leq \frac{q^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))-1}} + \frac{2q^2}{2^{s-1}} \leq \frac{3q^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))-1}} \end{aligned} \tag{3}$$

GAME_3. We now change the left oracle (i.e., the random oracle) such that the left oracle is always consistent with the right oracle. That is, instead of the random oracle we now give the distinguisher access to $H^{\mathcal{S}_0}$, that is, the iterative hash construction with the simulator $\mathcal{S}_0$ as oracle. Let GAME_3 be the event that distinguisher $\mathcal{D}$ outputs one in this setting, i.e.,

$$\Pr[\,\mathrm{GAME}_3\,] = \Pr\left[\,\mathcal{D}^{H^{\mathcal{S}_0},\mathcal{S}_0^{\mathcal{R}}}(1^\lambda) = 1\,\right] .$$

We will show that a distinguisher can only detect a difference in the view of GAME_2 and GAME_3 if the simulator $\mathcal{S}_0$ fails in at least one of the two games. In other words we show that in GAME_2 the responses of the simulator are always consistent with the random oracle, unless it explicitly fails.

For this note that the simulator uses the exact technique of the extractor from Lemma 3.4 to build its internal view. Furthermore, failure conditions B_1 to B_3 imply that the extraction properly succeeds in case the simulator sees all relevant queries. That is, collisions do not occur (B_1), the answer of the simulator is consistent with the random oracle on result queries (B_2) and that later generated partial graphs cannot be combined with earlier generated partial graphs (B_3). Thus, the only chance to not be consistent in game GAME_5 is, if the distinguisher manages to come up with the result of an intermediary query without querying the simulator and

[7] Note that the condition $\mathtt{hmp}^{-1}(m') = \mathtt{pg}.\mathrm{y}$ individually is not sufficient to allow for a graph to be extended which is why we can ignore it here.

thus the simulator does not recognize a potential result query. This, however, directly translates to an adversary in Lemma B.3 which we can turn into an indifferentiabiltiy distinguisher.

We, thus, have that the view is already consistent in game GAME_5 and thus

$$|\Pr[\,\mathrm{GAME}_3\,] - \Pr[\,\mathrm{GAME}_2\,]| \leq \mathsf{Adv}^{\text{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}') + \frac{4q^2}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))}}$$

GAME_4. In GAME_4 we make the game independent of the random oracle. That is, we now change the simulator to always choose output values for new queries as a uniformly random string in $\{0,1\}^s$. Furthermore, we remove any failure conditions from the simulator. This yields simulator $\mathcal{S}_1$ which effectively simulates a fixed length random oracle via lazy sampling. Let GAME_4 be the event that distinguisher $\mathcal{D}$ outputs one in this setting, i.e.,

$$\Pr[\,\mathrm{GAME}_4\,] = \Pr\left[\,\mathcal{D}^{H^{\mathcal{S}_1},\mathcal{S}_1}(1^\lambda) = 1\,\right] .$$

It holds that a distinguisher can only differentiate between games GAME_3 and GAME_4 if

- In game GAME_3, simulator $\mathcal{S}_0$ explicitly fails
- In game GAME_4, simulator $\mathcal{S}_1$ reaches a state in which simulator $\mathcal{S}_0$ would have explicitly failed.

For failure conditions we need to take conditions B_1 to B_3 into account. Putting it all together we have that

$$\begin{aligned}|\Pr[\,\mathrm{GAME}_4\,] - \Pr[\,\mathrm{GAME}_3\,]| &\leq \Pr[\,\mathcal{S}_3 \text{ fails in } \mathrm{GAME}_3\,] + \Pr[\,\mathcal{S}_4 \text{ reaches a failure condition}\,] \\ &= \frac{6q^2}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))-1}}\end{aligned}$$

We can now complete the proof of the theorem as we have reached the target view. Note that simulator $\mathcal{S}_1$ effectively implements a fixed length random oracle. Thus:

$$\left|\Pr\left[\,\mathcal{D}^{H^{\mathtt{h}},\mathtt{h}}(1^\lambda) = 1\,\right] - \Pr\left[\,\mathcal{D}^{H^{\mathcal{S}_1},\mathcal{S}_1}(1^\lambda) = 1\,\right]\right| = 0$$

and hence, summing up the probability loss in the various game steps we get

$$\mathsf{Adv}^{\text{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}_*}(\mathcal{D}) \leq \mathsf{Adv}^{\text{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}') + \frac{11q^2}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s))-1}} .$$

□

C.2 Derandomizing the Generic Simulator

We now show how the generic simulator constructed in the previous section can be derandomized. Note that the simulator distinguishes between result queries and non result queries (cp. lines 12 and 14 in Figure 19). Result queries are answered using the underlying simulator. As this simulator is probabilistic we need to define how randomness is generated for it. Likewise, we need to define how random values are chosen deterministically for non-result queries (cp. line 14 in Figure 19). Note that for non-result queries, the response is independent of gathered state, that is, it only depends on the query and the random coins of the simulator.

For the derandomization we rely on techniques developed by Bennet and Gill [BG81] who show that relatively to a random oracle $\mathcal{BPP}$ and $\mathcal{P}$ are identical.

Lemma C.3. *Let $H^{\mathtt{h}} : \{0,1\}^* \to \{0,1\}^n$ be a iterative hash function and $\mathcal{R}$ a random oracle. Let simulator $\mathcal{S}_*$ be constructed as in Construction C.1 and let it export the same interface as $\mathtt{h} : \{0,1\}^k \times \{0,1\}^d \to \{0,1\}^s$. Fix $t_{\mathcal{D}} \in \mathbb{N}$. Then there exists an efficient and* deterministic *simulator $\mathcal{S}_d$ such that for any distinguisher $\mathcal{D}$ with run-time bounded by $t_{\mathcal{D}}$ it holds that*

$$\Pr\left[\,\mathcal{D}^{\mathcal{R},{\mathcal{S}_d}^{\mathcal{R}}}(1^\lambda) = 1\,\right] = \Pr\left[\,\mathcal{D}^{\mathcal{R},\mathcal{S}_*^{\mathcal{R}}}(1^\lambda) = 1\,\right] .$$

Proof. We construct deterministic simulator $\mathcal{S}_d$ as follows. Simulator $\mathcal{S}_d$ works exactly as simulator $\mathcal{S}_*$ except for the generation of outputs in lines 12 and 14 in Figure 19. For answering a result query (m, x) (line 12) we generate the randomness for choosing the preimage of $\mathtt{g}^{-1}(\mathcal{R}(\mathtt{extract}(\mathtt{pg}))$ deterministically as

$$\mathcal{R}(1^{t_{\mathcal{D}}+1}\|m\|x)\|\mathcal{R}(1^{t_{\mathcal{D}}+2}\|m\|x)\|\ldots$$

picking the i-th bit of the stream as the i-th random bit.

For answering a non-result query (m, x) (line 14 in Figure 19) we compute the answer as

$$\mathcal{R}(0^{t_{\mathcal{D}}+1}\|m\|x)\ .$$

If we denote with R the random variable, mapping to the random bits used by simulator $\mathcal{S}_*$ and by $R_d^{\mathcal{R}}$ the random variable, mapping to the coins used by deterministic simulator $\mathcal{S}_d$ (over the choice of random oracle) then their statistical distance (denoted by $\delta(\cdot,\cdot)$) is zero, that is:

$$\delta(R, R_d^{\mathcal{R}}) := \frac{1}{2}\sum_x \left|\Pr[\,R = x\,] - \Pr_{\mathcal{R}}\left[\,R_d^{\mathcal{R}} = x\,\right]\right| = 0$$

As furthermore the queries to generate the random bits are larger than any queries made by any distinguisher with run time bounded by $t_{\mathcal{D}}$ it follows that

$$\Pr\left[\,\mathcal{D}^{\mathcal{R},{\mathcal{S}_d}^{\mathcal{R}}}(1^\lambda) = 1\,\right] = \Pr\left[\,\mathcal{D}^{\mathcal{R},\mathcal{S}^{\mathcal{R}}}(1^\lambda) = 1\,\right]$$

□

The simulator constructed in Lemma C.3 is not only deterministic, but it is also stateless with respect to non-result queries. That is, any non-result query (m, x) will be answered as $\mathcal{R}(0^{t_{\mathcal{D}}+1}\|m\|x)$ which is independent of any gathered state.

C.3 Proof of the Composition Theorem for UNSPLITTABLE Games: Theorem 4.2

With Lemma C.3 and C.2 we can now prove Theorem 4.2. For this, let us restate Theorem 4.2 from page 12, this time in a concrete setting.

Theorem 4.2. *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 $(t_{\mathcal{A}^*}, q_{\mathcal{A}^*}, \epsilon_G, \epsilon_{\mathsf{bad}})$-*UNSPLITTABLE *for $H^{\mathtt{h}}$ and let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be an adversary. Then, for any indifferentiability simulator $\mathcal{S}$ there exists adversary $\mathcal{B}_1, \ldots, \mathcal{B}_m$ and distinguisher $\mathcal{D}$ such that for all values y*

$$\Pr\left[\,G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m} \Rightarrow y\,\right] \le \Pr\left[\,G^{\mathcal{R},\mathcal{B}_1,\ldots,\mathcal{B}_m} \Rightarrow y\,\right] + \epsilon_G + \epsilon_{\mathsf{bad}} + \mathsf{Adv}^{\mathit{indiff}}_{H^{\mathtt{h}},\mathcal{R},\mathcal{S}}(\mathcal{D}') + \frac{11q_{\mathcal{D}}^2}{2^{\mathrm{H}_\infty(\mathtt{g}(U_s))-1}}$$

with

$$q_{\mathcal{D}} \le q_{G,0} + \sum_{i=1}^{m} q_{G,i} \cdot q_{\mathcal{A}_i^*}\ .$$

Values $q_{G,0}$ and $q_{G,i}$ denote upper bounds on the number of queries by game G to the honest interface of the hash functionality and to the i-th adversarial procedure, respectively.

Proof. The proof of our result almost directly follows with Theorem 6.1 in [RSS11a] (Theorem 4 in the proceedings version [RSS11b]), that is, the composition theorem by RSS for reset indifferentiability.

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 probability ϵ_{bad}, that is,

$$\left|\Pr\left[\,G^{H^{\mathtt{h}},\mathcal{A}_1,\ldots,\mathcal{A}_m}\,\right] - \Pr\left[\,G^{H^{\mathtt{h}},\mathcal{A}_1^*,\ldots,\mathcal{A}_m^*}\,\right]\right| \le \epsilon_G\ .$$

The RSS composition theorem for reset-indifferentiability tells us that for adversary $\mathcal{A}_1^*, \ldots, \mathcal{A}_m^*$ and simulator $\mathcal{S}$ there exists an adversary $\mathcal{B}_1, \ldots, \mathcal{B}_m$ and distinguisher $\mathcal{D}$ such that

$$\Pr\left[G^{H^{\mathsf{h}}, \mathcal{A}_1^*, \ldots, \mathcal{A}_m^*} \right] \leq \Pr\left[G^{\mathcal{R}, \mathcal{B}_1, \ldots, \mathcal{B}_m} \right] + \mathsf{Adv}^{\text{reset-indiff}}_{H^{\mathsf{h}}, \mathcal{R}, \mathcal{S}}(\mathcal{D}) \,.$$

Adversary $\mathcal{B}_i$ is defined as $\mathcal{B}_i^{\mathcal{R}} := \mathcal{A}_i^{*\mathcal{S}^{\mathcal{R}}}$, that is a separate instance of simulator $\mathcal{S}$ is used in each procedure $\mathcal{B}_i$. Distinguisher $\mathcal{D}^{\mathcal{F}.\text{hon}, \mathcal{F}.\text{adv}}$ is defined as $\mathcal{D} := G^{\mathcal{F}, \mathcal{A}_1, \ldots, \mathcal{A}_m}$ such that a reset call precedes any adversarial procedure call $\mathcal{A}_i^*$ [RSS11b]. For the remainder of the proof we show how to construct a simulator that is able to handle the reset calls made by $\mathcal{D}$.

SIMULATOR CONSTRUCTION. Let $\mathcal{S}_*$ be the simulator given by construction C.1. We derandomize $\mathcal{S}_*$ with Lemma C.3 and call the resulting simulator $\mathcal{S}_d$. For Lemma C.3 we need additionally to specify the runtime-bound. We use the runtime-bound of the above defined RSS distinguisher $\mathcal{D}$ and denote it by $t_{\mathcal{D}}$.

As described, adversary $\mathcal{B}_i$ in the RSS composition theorem is defined as $\mathcal{B}_i := \mathcal{A}_i^{*\mathcal{S}_d^{(i)}}$ where $\mathcal{S}_d^{(i)}$ denotes an independent copy of simulator $\mathcal{S}_d$. By construction, we know that for all non-result queries the simulation is guaranteed to be consistent, as independent instances of simulator $\mathcal{S}_d$ give the same answer to the same query. Further, by construction, during game $G^{\mathcal{R}, \mathcal{B}_1, \ldots, \mathcal{B}_m}$ bad result queries happen only with probability at most ϵ_{bad}. Thus, for result queries, all queries in the corresponding partial graph are with probability at least $1 - \epsilon_{\mathsf{bad}}$ queried by the same adversarial procedure in correct order. As this allows the procedure's copy of the simulator $\mathcal{S}_d$ to recognize the partial graph, it follows that also result queries can be answered consistently by the respective copy of simulator $\mathcal{S}_d$. Thus, with Lemma C.2 (the construction of simulator $\mathcal{S}_*$) and the fact that the advantage is not changed by the derandomization, we have that there exists distinguisher $\mathcal{D}'$ such that

$$\mathsf{Adv}^{\text{reset-indiff}}_{H^{\mathsf{h}}, \mathcal{R}, \mathcal{S}}(\mathcal{D}) \leq \epsilon_{\mathsf{bad}} + \mathsf{Adv}^{\text{indiff}}_{H^{\mathsf{h}}, \mathcal{R}, \mathcal{S}}(\mathcal{D}') + \frac{11 q_{\mathcal{D}}^2}{2^{\mathrm{H}_\infty(\mathsf{g}(U_s)) - 1}}$$

where the factor ϵ_{bad} is due to the probability of bad result queries.

□

game $\text{IK-CPA}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}(1^\lambda)$

$b \leftarrow \{0,1\}$
$(pk_0, sk_0) \leftarrow \mathsf{KGen}(1^\lambda)$
$(pk_1, sk_1) \leftarrow \mathsf{KGen}(1^\lambda)$
$m, st \leftarrow \mathcal{A}_1(pk_0, pk_1)$
$c \leftarrow \mathcal{E}^{H^{\mathfrak{h}}}(pk_b, m)$
$b' \leftarrow \mathcal{A}_2^{\mathfrak{h}}(c, st)$
return $(b = b')$

Figure 20: Key Indistinguishability under Chosen-Plaintext Attack

D Public-Key Extractability (PK-EXT) for PKE Schemes

Bellare et al. [BBDP01] define the notion of key indistinguishability (IK-CPA, see Figure 20) for PKE schemes which intuitively captures that no adversary can tell with which key, out of a known set, a ciphertext was encrypted. The notion is formalized as an indistinguishability experiment, where two keys are generated and given to the first-stage adversary $\mathcal{A}_1$ which outputs a target message. According to a secret bit b the message is encrypted with one of the two keys and the ciphertext is given to the second-stage adversary $\mathcal{A}_2$ which has to output a guess for b (note that as there is no restriction on the state shared by the two adversaries this is essentially a single-stage notion). The advantage of an adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ against IK-CPA is defined as

$$\mathsf{Adv}^{\text{IK-CPA}}_{\mathcal{AE}}(\mathcal{A}) := 2 \cdot \Pr\left[\text{IK-CPA}^{\mathcal{A}}_{\mathcal{AE}} \Rightarrow \mathsf{true}\right] - 1$$

It is easy to see that if the adversary can additionally choose the randomness of the scheme the notion cannot be fullfilled, as adversary $\mathcal{A}_2$ could then simply recompute the ciphertext for both keys and check which one it received.

In this section we show that our notion of PK-EXT-secure PKE schemes is fullfilled by the REwH1 scheme [BBN+09], if the underlying scheme is IK-CPA secure. Further, if the adversary cannot choose the randomness then IK-CPA implies PK-EXT for any PKE scheme.

RANDOMIZED-ENCRYPT-WITH-HASH. The Randomized-Encrypt-with-Hash (REwH1) scheme [BBN+09] builds on a PKE schem $\mathcal{AE}_r := (\mathsf{KGen}_r, \mathcal{E}_r, \mathcal{D}_r)$ in the random oracle model. The REwH1 scheme inherits key generation KGen and decryption $\mathcal{D}$ from $\mathcal{AE}_r$, while encryption is defined as

$$\mathcal{E}^{\mathcal{R}}(pk, m; r) := \mathcal{E}_r\left(pk, m; \mathcal{R}(pk \| m \| r)\right) .$$

D.1 Adaptive IK-CPA

For our result we need to adapt the IK-CPA notion such that the adversary can adaptively generate ciphertexts. Let us call the adaptive notion aIK-CPA. We depict the corresponding security game in Figure 21. As is the case for the standard IND-CPA notion for public key encryption, IK-CPA implies aIK-CPA. The proof follows from a standard hybrid argument.

Proposition D.1. *Let $\mathcal{A}$ be an aIK-CPA adversary making at most t queries to oracle* LoR*. Then there exists adversary $\mathcal{B}$ running in time of $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\textit{aIK-CPA}}_{\mathcal{AE}}(\mathcal{A}) \leq t \cdot \mathsf{Adv}^{\textit{IK-CPA}}_{\mathcal{AE}}(\mathcal{B})$$

We define $\text{IK-CPA0}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}$ exactly as $\text{IK-CPA}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}$ except that bit b is set to zero at the beginning of the game and the game returns the guess of adversary $\mathcal{A}_2$. Likewise, we define $\text{IK-CPA1}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}}$ where bit b is set to one. This allows us to write the advantage of an adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$ against IK-CPA as:

$$\mathsf{Adv}^{\text{IK-CPA}}_{\mathcal{AE}}(\mathcal{A}) := \Pr\left[\text{IK-CPA1}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}} \Rightarrow 1\right] - \Pr\left[\text{IK-CPA0}^{\mathcal{A}_1,\mathcal{A}_2}_{\mathcal{AE}} \Rightarrow 1\right]$$

game aIK-CPA$^{\mathcal{A}}_{\mathcal{AE}}(1^\lambda)$	**procedure** LoR(m)
$b \leftarrow \{0,1\}$	return $\mathcal{E}(pk_b, m)$
$(pk_0, sk_0) \leftarrow \mathsf{KGen}(1^\lambda)$	
$(pk_1, sk_1) \leftarrow \mathsf{KGen}(1^\lambda)$	
$b' \leftarrow \mathcal{A}^{\text{LoR}}(pk_1, pk_2)$	
return $(b = b')$	

Figure 21: Adaptive Key Indistinguishability under Chosen-Plaintext Attack

Similarly, we define games for the adaptive version:

$$\mathsf{Adv}^{\text{aIK-CPA}}_{\mathcal{AE}}(\mathcal{A}) := \Pr\left[\,\text{aIK-CPA1}^{\mathcal{A}}_{\mathcal{AE}} \Rightarrow 1\,\right] - \Pr\left[\,\text{aIK-CPA0}^{\mathcal{A}}_{\mathcal{AE}} \Rightarrow 1\,\right] \tag{4}$$

Proof of Proposition D.1. Let, without loss of generality, adversary $\mathcal{A}$ make exactly t queries. We define a sequence of adversaries $\mathcal{B}^i := (\mathcal{B}^i_1, \mathcal{B}^i_2)$ (for $0 < i \leq t$) against IK-CPA having access to an adversary $\mathcal{A}$ against aIK-CPA. Adversary $\mathcal{B}^i_1$ gets as input two public keys pk_0, pk_1. It simulates oracle LoR as follows: for the first $i-1$ queries m it answers with $\mathcal{E}(pk_0, m)$. For the i-th query m, adversary $\mathcal{B}^i_1$ simply outputs m together with its state. Adversary $\mathcal{B}^i_2$ receives as input ciphertext c which is either $\mathcal{E}(pk_0, m)$ or $\mathcal{E}(pk_1, m)$ and the state. It returns c as answer to the LoR-query. Adversary $\mathcal{B}^i_2$ answers all further LoR-queries m from adversary $\mathcal{A}$ with $\mathcal{E}(pk_1, m)$ and outputs as guess for bit b whatever adversary $\mathcal{A}$ outputs.

If b equals zero, then adversary $\mathcal{B}^t$ perfectly simulates the LoR oracle and likewise if b equals 1 then adversary $\mathcal{B}^0$ perfectly simulates the LoR oracle. Let $\mathcal{B} := (\mathcal{B}_1, \mathcal{B}_2)$ be the adversary that chooses $0 < i \leq t$ uniformly at random to then implement adversary $\mathcal{B}^i$. Thus, we have that

$$\begin{aligned}\Pr[\,\mathcal{B}\text{ outputs }0|b=0\,] &:= \sum_{j=1}^{t} \Pr\left[\,\mathcal{B}^j\text{ outputs }0|b=0 \wedge i=j\,\right] \cdot \Pr[\,i=j\,] \\ &= \frac{1}{t}\sum_{j=1}^{t} \Pr\left[\,\mathcal{A}^j\text{ outputs }0\,\right] \end{aligned} \tag{5}$$

where $\mathcal{A}^j$ is the adversary in the adaptive aIK-CPA game getting an LoR-oracle that on the first j queries uses public key pk_0 and on the remaining queries uses public key pk_1. Likewise, we have that

$$\begin{aligned}\Pr[\,\mathcal{B}\text{ outputs }1|b=1\,] &:= \sum_{j=1}^{t} \Pr\left[\,\mathcal{B}^j\text{ outputs }1|b=1 \wedge i=j\,\right] \cdot \Pr[\,i=j\,] \\ &= \frac{1}{t}\sum_{j=0}^{t-1} \Pr\left[\,\mathcal{A}^j\text{ outputs }1\,\right] \end{aligned} \tag{6}$$

Putting it all together, we have that

$$\begin{aligned}\mathsf{Adv}^{\text{IK-CPA}}_{\mathcal{AE}}(\mathcal{B}) &= \Pr\left[\,\text{IK-CPA1}^{\mathcal{B}_1,\mathcal{B}_2}_{\mathcal{AE}} \Rightarrow 1\,\right] - \Pr\left[\,\text{IK-CPA0}^{\mathcal{B}_1,\mathcal{B}_2}_{\mathcal{AE}} \Rightarrow 1\,\right] \\ &= \Pr\left[\,\text{IK-CPA1}^{\mathcal{B}_1,\mathcal{B}_2}_{\mathcal{AE}} \Rightarrow 1\,\right] - 1 + \Pr\left[\,\text{IK-CPA0}^{\mathcal{B}_1,\mathcal{B}_2}_{\mathcal{AE}} \Rightarrow 0\,\right]\end{aligned}$$

With equations (5) and (6) this yields

$$
\begin{aligned}
&= \Pr[\,\mathcal{B} \text{ outputs } 1 | b = 1\,] + \Pr[\,\mathcal{B} \text{ outputs } 0 | b = 0\,] - 1 \\
&= \frac{1}{t}\left(\sum_{j=0}^{t-1} \Pr\left[\,\mathcal{A}^j \text{ outputs } 1\,\right] + \sum_{j=1}^{t} \Pr\left[\,\mathcal{A}^j \text{ outputs } 0\,\right]\right) - 1 \\
&= \frac{1}{t}\left(\Pr\left[\,\mathcal{A}^0 \text{ outputs } 1\,\right] + \Pr\left[\,\mathcal{A}^t \text{ outputs } 0\,\right]\right) + \\
&\quad \frac{1}{t}\sum_{j=1}^{t-1}\left(\Pr\left[\,\mathcal{A}^j \text{ outputs } 1\,\right] + \Pr\left[\,\mathcal{A}^j \text{ outputs } 0\,\right]\right) - 1
\end{aligned}
$$

As $\left(\Pr\left[\,\mathcal{A}^j \text{ outputs } 1\,\right] + \Pr\left[\,\mathcal{A}^j \text{ outputs } 0\,\right]\right) = 1$ for all $1 \leq j \leq t-1$ this is

$$
\begin{aligned}
&= \frac{1}{t}\left(\Pr[\,\text{aIK-CPA1} \Rightarrow 1\,] + \Pr[\,\text{aIK-CPA0} \Rightarrow 0\,]\right) - \frac{1}{t} \\
&= \frac{1}{t}\left(\Pr[\,\text{aIK-CPA1} \Rightarrow 1\,] - \Pr[\,\text{aIK-CPA0} \Rightarrow 1\,] + 1\right) - \frac{1}{t}
\end{aligned}
$$

Finally, with equation (4) we get the advantage statement of the theorem:

$$= \frac{1}{t}\mathsf{Adv}^{\text{aIK-CPA}}_{\mathcal{AE}}(\mathcal{A})$$

which concludes the proof. □

D.2 REwH1 with IK-CPA implies PK-EXT

We can now show that the Randomized-Encrypt-with-Hash scheme is PK-EXT-secure if the underlying PKE scheme is IK-CPA-secure. We only consider the PK-EXT-notion in the random oracle model. Note that, as it is a single-stage notion this suffices for composition in the MRH theorem. Remember that $\mathsf{maxpk}_{\mathcal{AE}}$ denotes the maximum probability of a collision for a public-key as generated by KGen, defined in equation (2).

Theorem D.2. *Let $\mathcal{A}$ be a PK-EXT adversary making at most $q_{\mathcal{A}}$ random oracle queries. Then there exists an adversary $\mathcal{B}$ running in time of $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\textit{PK-EXT}}_{\textit{REwH1}}(\mathcal{A}) \leq \mathsf{Adv}^{\textit{aIK-CPA}}_{\mathcal{AE}}(\mathcal{B}) + (q_{\mathcal{A}} + 1) \cdot \mathsf{maxpk}_{\mathcal{AE}} \ .$$

Proof. We assume, without loss of generality, that adversary $\mathcal{A}$ does not repeat queries to its oracles.

We define adversary $\mathcal{B}$ against aIK-CPA. Adversary $\mathcal{B}$ gets as input two public keys pk_0 and pk_1 and runs adversary $\mathcal{A}$ against PK-EXT. It simulates $\mathcal{A}$'s queries to the ENC oracle using its LoR-oracle simply ignoring the randomness. That is, if (m, r) is a query by $\mathcal{A}$ to the ENC-oracle, then $\mathcal{B}$ answers this as $\mathtt{LoR}(m)$. Let q be a query to the random oracle by $\mathcal{A}$. Before answering, $\mathcal{B}$ tests if the first bits of query q equal one of the public keys, that is, if

$$q_{|_{1,\ldots,|pk_0|}} = pk_0 \qquad \text{or if} \qquad q_{|_{1,\ldots,|pk_1|}} = pk_1 \ .$$

If this is the case, then $\mathcal{B}$ terminates $\mathcal{A}$ and outputs 0 if the first bits were equal to pk_0 and 1 otherwise. If the first bits did not equal either of the keys it responds with $\mathcal{R}(q)$. If adversary $\mathcal{A}$ terminates with guess pk' then adversary $\mathcal{B}$ outputs 0 if $pk' = pk_0$, it outputs 1 if $pk' = pk_1$, and else outputs a random bit.

Let the event bad1 be defined as $\mathcal{A}$ queries its random oracle on message $pk_{1-b}\|x$ where x is a some bit string, which in turn leads to $\mathcal{B}$ outputting a wrong guess for b. As no information about pk_{1-b} is leaked to adversary $\mathcal{A}$ we can bind the probability of bad1 via a union bound with

$$\Pr[\,\mathsf{bad1}\,] \leq q_{\mathcal{A}} \cdot \mathsf{maxpk}_{\mathcal{AE}}$$

where $q_{\mathcal{A}}$ denotes the number of random oracle queries by adversary $\mathcal{A}$.

Let the event bad2 be defined as $\mathcal{A}$ outputs guess $pk' = pk_{1-b}$. With the same argument this probability is bound by

$$\Pr[\mathsf{bad2}] \leq \mathsf{maxpk}_{\mathcal{AE}}$$

If events bad1 and bad2 do not occur then note that adversary $\mathcal{B}$ perfectly simulates the oracles that are expected by $\mathcal{A}$. Adversary $\mathcal{A}$ expects the encryption scheme to use randomness generated as $\mathcal{R}(pk\|m\|r)$. This means that as $\mathcal{A}$ never queries the random oracle on $pk\|m\|r$ (this is implied by $\neg$bad1), it must expect the scheme to use uniformly random coins. This is exactly what is done by the LoR oracle. In this case adversary $\mathcal{B}$ wins whenever $\mathcal{A}$ outputs a correct guess (or queries the random oracle on $pk_b\|x$). This concludes the proof. □

A simple corollary of the theorem is, that if the adversary in the PK-EXT game is not allowed to specify the randomness used by the encryption scheme, then PK-EXT is directly implied by IK-CPA.

Corollary D.3. *Let $\mathcal{A}$ be a PK-EXT adversary which is not allowed to specify the randomness used by the encryption scheme. Then there exists an adversary $\mathcal{B}$ running in time of $\mathcal{A}$, such that*

$$\mathsf{Adv}^{\textit{PK-EXT}}_{\mathcal{AE}}(\mathcal{A}) \leq \mathsf{Adv}^{\textit{aIK-CPA}}_{\mathcal{AE}}(\mathcal{B}) + (q_{\mathcal{A}} + 1) \cdot \mathsf{maxpk}_{\mathcal{AE}} \ .$$

E The Ideal Cipher Model vs. the Ideal Compression Function model

We have stated all of our results relatively to ideal compression functions $\mathtt{h} : \{0,1\}^k \times \{0,1\}^d \to \{0,1\}^s$. In this section we briefly discuss the relation between ideal compression functions and ideal ciphers, where an ideal cipher is a function chosen uniformly at random from all keyed permutations $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ where for each key $\kappa \in \{0,1\}^k$ function $E_\kappa(\cdot) := E(\kappa, \cdot)$ defines a random permutation.

Compression functions in hash functions are often build from keyed permutations, for example, using the Davies-Meyer (DM) construction [Win83], which relative to keyed permutation $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ defines a compression function as $\mathtt{h}(m,x) := E_m(x) \oplus x$. We give the schematic of the construction in Figure 22.

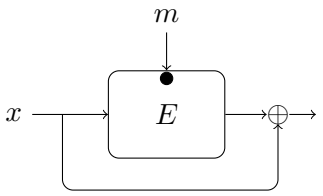

Figure 22: The Davies-Meyer construction [Win83] of a compression function from an (ideal) keyed permutation E.

Now it is easily seen, that the DM construction is not an ideal compression function, since given values $\mathtt{h}(m,x) \oplus x$ and y it is possible to reconstruct value $x = E_m^{-1}(h(m,x) \oplus x)$, which for an ideal compression function should not be possible [CDMP05]. We do, however, expect that the DM construction can be plugged into all our statements which are then analyzed in the ideal cipher setting. Note that we did not formally verify this claim. In the following we do, however, give our reasoning: As discussed also for the sponge construction (see Section B.6), our proofs make use of three properties of compression functions:

1. To argue that hash values cannot be learned without querying all $\mathtt{h}$-queries that also occur within an honest execution (see the missing link lemma; Lemma 3.3), we require that over the choice of $\mathtt{h}$ the random variable $\mathtt{h}(m,x)$, for fixed m and x has high min-entropy. For the DM construction this is the case as over the choice of E value $E_m(x) \oplus x$ can take any value in $\{0,1\}^n$.

2. We require that compression function $\mathtt{h}$ is collision resistant in order to rule out that two execution graphs labeled relative to $\mathtt{h}$ have a common subgraph which includes the final node $\mathtt{g}$ (see the extractor lemma; Lemma 3.4 as well as Lemma C.2). Dodis et al. [DRS09] show that the DM construction is preimage aware (PrA) which implies collision resistance.

3. We require that a hash value $H^{\mathtt{h}}(M)$ for a message M does not leak any more information on queries to $\mathtt{h}$ during the computation of $H^{\mathtt{h}}(M)$ than does message M (see the proof of Theorem 5.4). This can be interpreted as, given value $H^{\mathtt{h}}(M)$ the strengthened missing link lemma still holds (see Lemma B.2). Hash values are computed as $\mathtt{g}(\mathtt{h}(m,x))$ for a result query (m,x) where x is a chaining value and which translates to $\mathtt{g}(E_m(x) \oplus x)$ for the DM construction. As E is an ideal cipher, value $E_m(x) \oplus x$ is uniformly distributed in $\{0,1\}^s$. As x is a chaining value and by (1) and the strengthened missing link lemma has full entropy, value x cannot be learned but for a local re-computation of $H^{\mathtt{h}}$ (M), or via exhaustive search on E and E^{-1}.

Let us note that if hash function $H^{\mathtt{h}}$ is required to be indifferentiable from a random oracle, that then this naturally translates to H^{DM^E} needing to be indifferentiable from a random oracle. For the case of HMAC, NMAC, and various Merkle-Damgård variants including chopMD and prefix-free-MD Coron et al. [CDMP05] have given analyses in precisely this setting.