
Proceedings Paper:
Dodd, Charles John, Farshim, Pooya, Shahandashti, Siamak F. et al. (2026) Multi-Instance Unrecoverability of iMHF-Based Password Hashing. In: 31st European Symposium on Research in Computer Security. 31st European Symposium on Research in Computer Security, 14-18 Sep 2026 , ITA. (In Press)

Multi-Instance Unrecoverability of iMHF-Based Password Hashing

Charles Dodd[1,3], Pooya Farshim[1,2], Siamak F. Shahandashti[3], and Karl Southern[1]

[1] Department of Computer Science, Durham University, Durham, UK
{charles.j.dodd,pooya.farshim,karl.southern}@durham.ac.uk
[2] IOG, Switzerland
[3] Department of Computer Science, University of York, York, UK
siamak.shahandashti@york.ac.uk

Keywords: password hashing · memory-hard function · multi-instance security · cumulative memory complexity · pebbling complexity

Abstract. The study of memory-hard functions (MHFs) so far has mainly focused on providing provable guarantees on the expected minimum cumulative memory complexity (CMC) required per *evaluation* when amortized over multiple instances. Such results, however, do not provide any guarantees for the security of compromised password banks in the sense of passwords remaining *unrecoverable*. Indeed, a construction can be memory-hard while still leaking information about its input. We provide the first formal treatment of the unrecoverability of graph-based data-independent MHFs (iMHFs) in the multi-instance setting. Multi-instance security is the accepted security model when inputs have low-entropy or are correlated, and require the adversarial effort to linearly scale with the number of instances broken.

To prove these results, we appropriately extend the ex-post-facto pebbling technique of Alwen and Serbinenko (STOC'15) and the unguessability reductions of Farshim and Tessaro (EUROCRYPT'21). We then use the resulting compatible frameworks to bound the number of guesses of adversaries with a given CMC in terms of the pebbling complexity of the graph underlying the iMHF. Combined with known lower bounds for the pebbling complexities of their graphs, we obtain, as corollaries, concrete unrecoverability bounds for the Argon2i, Catena, and Balloon hashing, showing in particular that the advantage indeed scales linearly with the number of instances and the cumulative memory complexity of the attacker.

1 Introduction

Password-based cryptography leverages the prevalence of passwords to underpin multiple applications such as authentication and key derivation. Password storage in such applications is usually protected through salting and hashing, i.e., rather than a password pw in plaintext, a pair $(sa, \mathsf{H}(pw||sa))$ is stored for

a random salt sa. To slow down password-cracking attacks, H is replaced with a construction C^{H} built from H through *iteration*, as in the case of bcrypt [14] and PBKDF2 [13]. These constructions are designed to make bulk computations prohibitively costly, while incurring only a small cost on authenticating honest users. However, the ever-growing efficiency of specialized hardware such as ASICs in evaluating hash functions necessitates higher iteration counts that disproportionately affect honest users.

To address this issue, recommendations for password hashing [1] advocate the use of *memory-hard functions* (MHFs) designed to ensure that successful computation of the function requires expending large amounts of time and memory. This leverages the fact that dedicated hardware does not provide the same cost saving in scaling up memory as in computation. Candidates for MHFs include Scrypt [12], the first proposal for such a function in 2009 by Percival, and Argon2 [6] chosen through the Password Hashing Competition in 2015 [1].

MHFs come in two flavors, distinguished by their memory access pattern. In *data-dependent* MHFs (dMHFs), the memory access pattern changes with the input, whereas *data-independent* MHFs (iMHFs) have the same memory-access pattern for every input. The latter is important when the function is run on a secret input (such as passwords) in an insecure environment, as dMHF memory-access patterns can potentially allow the recovery of secrets/passwords via side-channel attacks [8]. Constructions of iMHFs are usually based on an underlying *directed acyclic graph* (DAG) that specifies how hash functions are combined to give the construction. Such iMHFs are called *graph-based* iMHFs. Examples include Argon2i [6], Catena [10], and Balloon hashing [7].

Multi-instance security. In practice, passwords may be weak, and even if stored under iterated hashing or an MHF C^{H}, they are still vulnerable to guessing attacks: an adversary with access to a leaked password database $(sa_i, \mathsf{C}^{\mathsf{H}}(pw_i||sa_i))$ for passwords pw_i, can attempt to recover them by guessing common passwords pw^*, computing $\mathsf{C}^{\mathsf{H}}(pw^*||sa_i)$ for each pw^*, and comparing it with the database entries. Hence, there is an inherent limit to the protection that constructions can provide for *individual* passwords. Despite this, one still would like assurances that the effort needed to recover *multiple* passwords increases, e.g., *scales linearly*, with the number of passwords recovered. The *multi-instance* (mi) security model, introduced by Bellare, Ristenpart, and Tessaro (BRT12) [4], formalizes this metric based on how the resources expended by the adversary grow with the number of target hashes that are "cracked". This model was further studied by Farshim and Tessaro (FT21) [9] who proved the efficacy of salting against preprocessing attacks (e.g., rainbow tables [11]) in the multi-instance setting.

In light of ASIC-assisted attacks, measuring the complexity of attacks through query complexity is not sufficient. Percival [12] proposed maximum memory usage multiplied by time as an alternative metric. Subsequently, this metric was

shown to provide insufficient guarantees against *amortization* [3].[4] The state-of-the-art metric for measuring memory cost, proposed by Alwen and Serbinenko (AS15) [3], is the *cumulative memory complexity* (CMC): the *sum* of memory usage at each time step in the computation. Alwen and Serbinenko prove lower bounds on the CMC of graph-based iMHFs based on the *cumulative complexity* (CC) of the underlying graph, defined as the minimum *pebbling* complexity of the graph [3]. This allows them to prove the non-amortizability of MHF evaluations.

Although provable guarantees on the memory-hardness of graph-based iMHFs provide high confidence that *computation* would incur a certain minimum memory cost, they do not translate into provable *unrecoverability* guarantees for password storage using such iMHFs. To see this, given a well-designed MHF F^{H}, consider the construction $\mathsf{C}^{\mathsf{H}}(x) := \mathsf{F}^{\mathsf{H}}(x)||x$. This construction is memory-hard, as computing the output requires computing F^{H}. However, it does not provide any security for password storage as *recovering* the input is trivial. More generally, a construction $\mathsf{F}_1^{\mathsf{H}}(x)||\mathsf{F}_2^{\mathsf{H}}(x)$ will have a CMC of at least the maximum CMC of $\mathsf{F}_1^{\mathsf{H}}$ and $\mathsf{F}_2^{\mathsf{H}}$, but will only achieve an unrecoverability bound of at most the *minimum* unrecoverability bounds that $\mathsf{F}_1^{\mathsf{H}}$ and $\mathsf{F}_2^{\mathsf{H}}$ achieve.

To date, there has been no formal unrecoverability analysis of MHFs in the literature. However, the natural security property expected from password storage is that given a leak of the form $(sa_i, \mathsf{C}^{\mathsf{H}}(pw_i||sa_i))$, it is hard to *recover* a significant number (i.e., multiple instances) of the original passwords given certain adversarial resources.

Our contributions. Motivated by the prevalence of MHFs in password-based protocols, we provide the first formal treatment of multi-instance unrecoverability of graph-based iMHFs. We prove concrete upper bounds on the unrecoverability of passwords based on the unguessability of the underlying password distribution, the CMC of the adversary, and the CC of the graph underlying an iMHF. Towards this, we show how to combine the FT21 and AS15 frameworks to derive unrecoverability bounds for iMHFs. Our proofs bound the number of guesses of any adversary with a given CMC in terms of the CC of the underlying graph via an extension of the ex-post-facto pebbling argument of AS15. It then applies the unrecoverability-to-unguessability reduction of FT21 to derive our final bounds.

Along the way, we give a definition of *strong* memory-hardness (sMH), whereby an adversary may choose its inputs to the iMHF adaptively based on the iMHF's underlying hash function. This definition is stronger than that due to AS15: an MHF with high (even optimal) CMC can have fast evaluations, i.e., have a non-optimal strong memory hardness. For example whilst `Scrypt` is shown to be maximally memory hard [2], we show in Appendix A that this is not the case in the strong memory-hardness game. Still, our results show that the sMH of graph-based iMHFs can be tightly related to the CC of their underlying graphs.

[4] The maximum memory usage can be high even if it is high only at the initial stages of the attacks, thus allowing reuse of freed memory for a second instance after the initial stage is over.

2 Preliminaries

Notation. We denote the size of a set N by $|N|$, the length of a vector $\mathbf{x}$ by $|\mathbf{x}|$, and the set of its unique members by $\{\mathbf{x}\}$. For sampling a value v uniformly at random from a set N, we write $v \twoheadleftarrow N$. When sampling a value v randomly from a distribution $\mathcal{D}$ (or by a randomized algorithm), we write $v \twoheadleftarrow \mathcal{D}$. For the set of integers from 1 to m we write $[m]$. We denote the set of functions with domain N and range M as $\mathrm{Fun}(N, M)$. An adversary $\mathcal{A}$ is a randomized algorithm, and $\mathcal{A}^{\mathcal{O}}$ denotes that it has access to an oracle $\mathcal{O}$. We write ε for the empty string.

2.1 Basic security definitions

The parallel random oracle model. The random oracle model (ROM) [5] is an idealized model of computation in which all parties, honest or otherwise, have access to a uniformly sampled function $\mathsf{H} \twoheadleftarrow \mathrm{Fun}(N, M)$. We consider adversaries in the *parallel random oracle model* (pROM) [3]. In this model, an algorithm can make batches of queries to the random oracle H in rounds, and receive the corresponding responses simultaneously. More precisely, a pROM algorithm $\mathcal{A}_*^{\mathsf{H}}$ takes an input z, and makes t rounds of batch oracle calls. In each round, $\mathcal{A}$ makes a batch of queries $\mathbf{q}$ and writes to a free state st. The batch of responses $\mathbf{y}$ along with st are subsequently given to $\mathcal{A}$, and $\mathcal{A}$ writes to a state σ_i any information passed to the following round. $\mathcal{A}$ may also append values to a special output register $\sigma_{\mathcal{O}}$. To append some value l to the output register, we assume $\mathcal{A}$ makes a special query of the form (l, out), where out is a special symbol. This will not be answered by the oracle, but recorded in the transcript. When the algorithm terminates, the contents of $\sigma_{\mathcal{O}}$ are returned. See Fig. 1 (top left) for the structure of a pROM algorithm.

Password samplers. A password sampler $\mathcal{P}_m$ in BRT12/FT21 is defined to be a randomized algorithm that takes no input and outputs a vector of m passwords $\mathbf{pw}$ and possibly some leakage information z on the passwords. The leakage z models the information the adversary holds about the password distribution prior to its attack on a password-based cryptosystem.

Salt generators. Salts are generated by a salt-generator algorithm Gen. This is a randomized algorithm which takes as input a user index $i \in [0, m)$ and a counter $j \in [0, \ell)$, and outputs a salt $\mathbf{sa}[i, j]$ from a space of size K.

Unguessability and salted unguessability. We briefly recall the notions of (salted) unguessability from [4,9], the security games for both are given in Fig. 1. Intuitively, guessability measures the adversary's ability to recover a full bank of passwords, sampled from a password distribution, using some leakage z on the passwords. The most basic game to address this notion is the Guess game from [4]. An adversary $\mathcal{A}$ in the Guess game is given leakage z, with access to test and corruption oracles Test and Cor. Queries to Test allow $\mathcal{A}$ to check password guesses, returning a win_i flag if the ith password is guessed correctly.

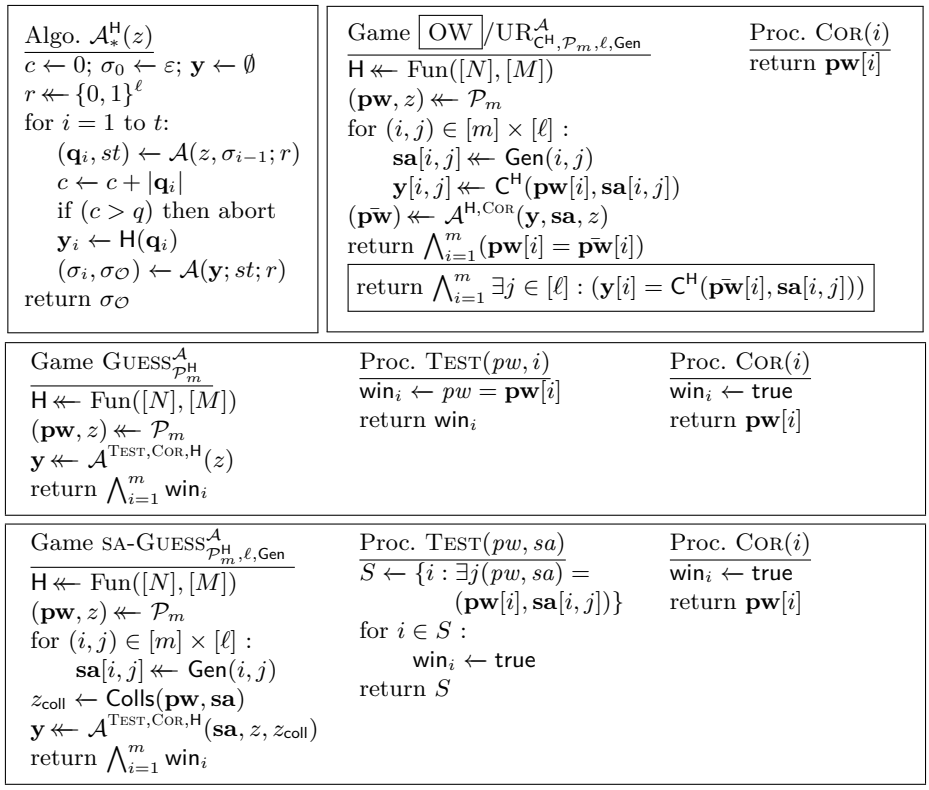

Fig. 1: Top left: Structure of an algorithm in pROM. Top right: UR and OW games w.r.t. a m-sampler $\mathcal{P}_m$ and salt generator Gen. Middle: Guess game w.r.t. $\mathcal{P}_m$. Bottom: sa-Guess game w.r.t. $\mathcal{P}_m$. Here z_{coll} denotes the collision pattern of password-salt pairs.

We consider adversaries that are able to make at most c queries to Cor, i.e., they automatically "win" on c of the passwords.

For the salted setting, we again consider adversaries that are able to make up to c corruption queries to Cor. The Test oracle now takes password-salt pairs as queries. Each query $\mathcal{A}$ sends to Test is checked against each password-salt pair in the sampled set, and win flags are returned for each matching password which has a matching salt. This models the fact that the adversary can verify the guess without repeating it for each salt. The collision pattern on password-salt pairs is given explicitly to the adversary by procedure $\mathsf{Colls}(\mathbf{pw}, \mathbf{sa})$. This takes the vector of m passwords and the $m \times \ell$ matrix of salts, and returns the password-salt collision pattern z_{coll}. This is an $m\ell \times m\ell$ matrix with entries $((i_1, j_1), (i_2, j_2))$ set to 1 only when $(\mathbf{pw}[i_1], \mathbf{sa}[i_1, j_1]) = (\mathbf{pw}[i_2], \mathbf{sa}[i_2, j_2])$.

Unrecoverability and one-wayness. We recall the notions of unrecoverability and one-wayness with relation to password hashing, the security games for both are given in Fig. 1. The unrecoverability of passwords models the hardness of recovering passwords given a password bank protected by a hash-based construction. We consider the unrecoverability game for a general hash-based construction C^{H}. In this game, a password sampler $\mathcal{P}_m$ outputs m passwords along with some leakage z. The salt generator Gen outputs the salt matrix $\mathbf{sa}$, which along with the password vector $\mathbf{pw}$ is used as input to C^{H}, generating the challenge vector $\mathbf{y}$. The adversary is given $\mathbf{y}$ along with $\mathbf{sa}$ and the leakage z. Note that for this game, the collision pattern is public and does not need to be provided to $\mathcal{A}$.

2.2 Graph-based iMHFs and cumulative memory complexity

Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG), where the node set V is the set of integers 1 to n, referred to as node indices. Let $\mathbf{Pa}(i)$ (resp. $\mathbf{Ch}(i)$) be the set of parents (resp. children) of the node i, with $\mathbf{Pa}_j(i)$ (resp. $\mathbf{Ch}_j(i)$) being the j-th parent (resp. child) ordered by node index. Nodes with no parents or children are called *source* and *sink* nodes, respectively. Let δ_i be in-degree of the ith node and $\delta := \max_i\{\delta_i\}$. We consider iMHFs which use the labeling scheme from AS15 [3]. We reproduce the definition of the labeling here, and extend it to allow for salting. To allow salting, we extend the initial label l to be drawn from $L \times K$ rather than L (i.e., l would take the form (pw, sa)).

(H, l)-labeling of graphs [3]. Let $\mathcal{G} = (V, E)$ be a DAG with maximum in-degree δ, L be an arbitrary label set, $[K]$ be a salt set of size K, and $\mathbb{H} := \mathrm{Fun}(V \times L^{\delta} \times [K], L)$. For a function $\mathsf{H} \in \mathbb{H}$ and a label $l \in L \times [K]$, the (H, l)-labeling of $\mathcal{G}$ is a mapping $\mathbf{lab} : V \mapsto L$ defined recursively for all v by:

$$\mathbf{lab}(v) := \begin{cases} \mathsf{H}(v, l) & : \mathrm{indeg}(v) = 0\,; \\ \mathsf{H}(v, \mathbf{lab}(\mathbf{Pa}_1(v)), \ldots, \mathbf{lab}(\mathbf{Pa}_d(v))) & : 0 < \mathrm{indeg}(v) = d \leq \delta\,. \end{cases}$$

There can be at most $\delta + 1$ inputs to H. We assume an injective padding function is applied to the inputs to give exactly $\delta + 2$ inputs before H is applied to compute the label. We, however, omit this padding function for clarity. We use $\mathbf{pre\text{-}lab}(v)$ to refer to the H inputs which define v's label, i.e., $\mathbf{pre\text{-}lab}(v) = (v, l)$ if $\mathrm{indeg}(v) = 0$, and $\mathbf{pre\text{-}lab}(v) := (v, \mathbf{lab}(\mathbf{Pa}_1(v)), \ldots, \mathbf{lab}(\mathbf{Pa}_d(v))$ otherwise. Note that as l is a password-salt pair, a salt is only included when $\mathrm{indeg}(v) = 0$.

Using this labeling definition, a graph-based iMHF C^{H} is defined as follows.

Graph-based iMHF [3]. Let $\mathcal{G} = (V, E)$ be a DAG on n vertices, with maximum in-degree δ, a single source node, and a single sink node v_s. Let L be an arbitrary label set, $[K]$ be the salt set, and $\mathbb{H} = \mathrm{Fun}(V \times L^{\delta} \times [K], L)$. The *graph functions* (of $\mathcal{G}$ and $\mathbb{H}$) are the members of the family of oracle functions $\mathsf{C} = \mathsf{C}^{\mathbb{H}}_{\mathcal{G}}$, indexed by functions in $\mathbb{H}$, mapping $L \times [K]$ to L. For input $l \in L \times [K]$, the value of $\mathsf{C}^{\mathsf{H}} \in \mathsf{C}$ is defined as $\mathsf{C}^{\mathsf{H}}(l) := (\mathbf{lab}(v_s))$, where $\mathbf{lab}$ is the (H, l)-labeling of $\mathcal{G}$.

We primarily consider the labeling set $L := [N]$, but the results hold for an arbitrary labeling set L where $|L| = N$ that can be efficiently mapped to $[N]$. We follow the notation used in [3] for denoting multiple instances of a graph. That is, for a DAG $\mathcal{G} = (V, E)$ and a positive integer m, we use $\mathcal{G}^{\times m}$ (which we call the tensor graph) to denote the disjoint union of m copies of $\mathcal{G}$, where each node has a unique index of the form (v, k) for $v \in V$ and $k \in [m]$. We denote constructions based on $\mathcal{G}^{\times m}$ and random oracle H by $\mathsf{C}^{\mathsf{H}}_{\times m}$. Note that $\mathsf{C}^{\mathsf{H}}_{\times m}$ is defined by a labeling on a graph with distinct node indices.

In order to model the memory requirements for an adversary to compute an iMHF, one usually analyzes an adversary playing the *(black) pebbling game* on the underlying graph. In the pebbling game, an adversary is challenged with placing pebbles on the sink nodes of a DAG, while following some simple rules as follows. The pebbling occurs in rounds. In each round, an adversary can add a single pebble and remove any number of pebbles from the graph. The source nodes for the graph can be pebbled at any time. For all other nodes, an adversary can only place a pebble on that node if all of its parents had pebbles on them in the previous round. We consider the *parallel* pebbling game where batches of pebbles can be added (or removed) in each round.

Parallel pebbling. Formally, given a DAG $\mathcal{G} = (V, E)$, the parallel (black) pebbling of $\mathcal{G}$ is defined as a sequence of configurations $P = (P_0, \ldots, P_t)$, where $P_i \subseteq V$. A pebbling with starting and target sets $S, T \subset V$ is *legal* if the following three conditions hold: (1) $P_0 = \emptyset$; (2) a node can only be added when its predecessors have a pebble on, or if it is a starting node, i.e., $\forall i \in [t], \forall x \in P_i \setminus P_{i-1} : (\forall y \in \mathbf{Pa}(x) : y \in P_{i-1}) \vee (x \in S)$, where there is no limit on the number of pebbles that can be placed in a configuration; and (3) all target nodes are pebbled at some round, i.e., $\forall x \in T,\ \exists z \leq t : x \in P_z$. A pebbling is *complete* if S and T are the set of source and sink nodes, respectively. N.B. that by (3) all nodes in T are pebbled at some point in time.

Cumulative pebbling complexity. Given a parallel pebbling adversary $\mathcal{A}$ that places at most q pebbles in total, across at most t rounds, and produces a pebbling, we define the cumulative pebbling complexity (CPC) of $\mathcal{A}$'s pebbling of $\mathcal{G}$ as

$$\mathrm{CPC}(\mathcal{A}) := \sum_{i=0}^{t-1} |P_i| \ .$$

We define the q-CPC of a graph $\mathcal{G}$ as the minimum CPC of any q-pebble adversary's legal and complete pebbling of $\mathcal{G}$, i.e., let $S[q]$ be the set of all q-pebble adversaries, then $\mathrm{CPC}_q(\mathcal{G}) := \min_{\mathcal{A} \in S[q]}(\mathrm{CPC}(\mathcal{A}))$. Above a sufficiently large q (e.g., if $|V| = n$, for $q > n^2$), $\mathrm{CPC}_q(\mathcal{A})$ will not decrease, as an adversary with more pebbles can always use a sufficient number of them. Hence, for sufficiently large q, CPC is independent of q. We define this minimum as the *cumulative complexity* of the graph $\mathrm{CC}(\mathcal{G})$.

Cumulative memory complexity. We consider memory-constrained adversaries in pROM, which make batches of queries to the random oracle in rounds. Keeping track of the state size (which measures the memory stored between rounds), allows for an analysis of the total memory used by a parallel adversary across its execution time. We define the cumulative memory complexity of $\mathcal{A}$ with respect to H, input x, and coins r as

$$\mathrm{CMC}(\mathcal{A}, x, \mathsf{H}, r) := \sum_{i=0}^{t} |\sigma_i| \ ,$$

where σ_i is the state after the ith round as defined in Section 2.1. For our applications, the input to $\mathcal{A}$ is an empty string and so we use $\mathrm{CMC}(\mathcal{A})$, leaving both r and H implicit. Following [3], we define the cumulative memory complexity of $\mathcal{A}$ with respect to x as

$$\mathrm{CMC}(\mathcal{A}, x) := \mathbb{E}_{\mathsf{H},r}\left[\mathrm{CMC}(\mathcal{A}, x, \mathsf{H}, r)\right] \ ,$$

where the expectation is taken over the choice of random oracle H and random coins r. Since our proofs are written with respect to an H, we use the former of these two equations, rather than needing to take the expectation over all H.

Single-sink vs. multi-sink graphs. Current metrics for CMC assume that a full computation of a construction has to be done, and that a partial computation is not useful to an adversary. For the purposes of guessing passwords, this assumption is false. Consider a function F^{H} with a single sink which has high CMC, and define a construction $\mathsf{C}^{\mathsf{H}} := \mathsf{F}^{\mathsf{H}}(x)||\mathsf{H}(x)$. Now, C^{H} still has a high CMC. However to check if $\mathsf{C}^{\mathsf{H}}(x) = \mathsf{C}^{\mathsf{H}}(x')$, i.e., to check if a guessed password is correct, an adversary only needs to make a single hash query: the adversary computes $\mathsf{H}(x')$ and if it doesn't match the second half of the output of $\mathsf{C}^{\mathsf{H}}(x)$, then $x \neq x'$. Note that this holds not only for unrecoverability, but even for finding collisions. Whilst the example we have given is slightly contrived, it is clear that for any multi-sink construction the CMC required to compute any sink could be much lower than the CMC required to compute all sinks. We therefore only consider constructions with a single sink. We note that major graph based iMHFs, e.g., Argon2i [6], Catena [10] and Balloon hashing [7], all adhere to this restriction.

3 Strong Memory Hardness

In existing hash-independent definition of memory-hard functions, a parallel adversary chooses the input x to the function at the beginning, and the function is deemed memory hard if the adversary requires a large amount of CMC to correctly evaluate the function (with high probability). Importantly, the cost of evaluations must not be amortizable across multiple instances. In this setting, the adversary is restricted to choosing the input to the function *independently* of the random oracle. The security game for the standard definition is presented as the m-MH game in Fig. 2 (left). This game is for two-stage adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$.

Game m-MH$^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}$	Game (q,m)-sMH$^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}$
$\mathsf{H} \twoheadleftarrow \mathrm{Fun}([N],[M])$	$\mathsf{H} \twoheadleftarrow \mathrm{Fun}([N],[M])$
$\mathbf{x} \twoheadleftarrow \mathcal{A}_0()$	$(\mathbf{x},\mathbf{y}) \twoheadleftarrow \mathcal{A}^{\mathsf{H}}()$
if $(\lvert\{\mathbf{x}\}\rvert \neq m)$:	if $(\lvert\{\mathbf{x}\}\rvert \geq q) \vee (\lvert\{\mathbf{y}\}\rvert \geq q)$:
return 0	return 0
$\mathbf{y} \twoheadleftarrow \mathcal{A}_1^{\mathsf{H}}(\mathbf{x})$	for x in $\mathbf{x}$ and y in $\mathbf{y}$:
return $(\mathbf{y} = \mathsf{C}^{\mathsf{H}}(\mathbf{x}))$	if $y = \mathsf{C}^{\mathsf{H}}(x)$:
	$S \leftarrow S \cup \{(x,y)\}$
	return $\lvert S\rvert \geq m$

Fig. 2: The hash-independent (left), and the strong (right) memory hardness games. In the unsalted setting, $\mathbf{x}$ is a vector of passwords pw_i. In the salted setting, $\mathbf{x}$ is a vector of tuples of the form (pw_i, sa_i).

With no access to H, $\mathcal{A}_0$ picks a vector $\mathbf{x}$ of m distinct inputs. Then $\mathcal{A}_1$, given $\mathbf{x}$ as input, makes batch queries to H, and outputs a vector of evaluations $\mathbf{y}$. Adversary $\mathcal{A}$ wins if for each input value x_i in $\mathbf{x}$, the corresponding output y_i in $\mathbf{y}$ satisfies $\mathsf{C}^{\mathsf{H}}(x_i) = y_i$. More precisely, the construction C^{H} is strongly (m, q, ϵ, μ)-memory hard if for all q-query adversaries $\mathcal{A}$ with CMC($\mathcal{A}$) at most μ, adversary $\mathcal{A}$ wins the m-MH game with probability at most ϵ.

We introduce a stronger notion, the *strengthened* MH game sMH, that extends adversarial capabilities in two orthogonal ways. First, it grants the adversary oracle access whilst choosing inputs. Second, it allows the adversary to output an *unordered* set of q inputs and q outputs, where only m are required to be correct MHF computations. This game is formalized in Fig. 2 (right). We define adversary $\mathcal{A}$'s advantage in the strong MH game as

$$\mathbf{Adv}^{(q,m)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) := \Pr[(q,m)\text{-sMH}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}] \ .$$

Strong (m, q, ϵ, μ)-memory hard is defined analogously.

We build on the results of AS15 and show that, in contrast to general MHFs, *graph-based* MHFs are indeed memory hard even if the input instances are picked in a hash-dependent way. To this end, we adapt the ex-post-facto pebbling argument. We first give a brief overview of how memory hardness is established in the hash-independent setting.

The hash-independent case. AS15 proves a lower bound on the amortized CMC of evaluating m instances of an iMHF based on a graph $\mathcal{G}$, with probability at least ϵ. The bound shows that the best adversary evaluating the function (with high probability) will have cumulative memory cost close to adversaries which simply pebble each instance of the graph. This result is proved by applying the results of three auxiliary lemmas to the definition of the amortized cost. The first ([3, Lemma 11]) calculates the expected adversarial savings made by using collisions between the labeling of the instances. This bounds the cost of computing m possibly colliding instances by the cost of fewer non-colliding instances. The m non-colliding instances are then treated as a single instance of a larger

construction based on the tensor graph $\mathcal{G}^{\times m}$, whose cumulative complexity is lower bounded in [3, Lemma 12]. The CMC of an adversary computing a single instance of this construction is then bounded, roughly, by the cumulative complexity of the underlying tensor graph. This is done by analyzing an adversary's queries, showing that a legal and complete pebbling of the underlying graph can be extracted from any correct evaluation. The third result ([3, Lemma 3]) proves the cumulative complexities of two graphs are additive. The bound for the one-off cost of the large graph can then be given in terms of that of the smaller graph underlying the construction. Together, in [3, Theorem 3], these lemmas bound the amortized cost in terms of the CC of the corresponding graph.

Extension to strong memory-hardness. To prove our bound, we extend the technique in [3, Lemma 12]. Note that by omitting the final step of [3, Lemma 12], in which the authors derive a lower bound on the cost, we can obtain an upper bound on the advantage. Extending the first steps of the AS15 proof, we give a reduction from an adversary in the (q, m)-sMH game to a prediction game, in which an adversary predicts the output of the random oracle on an unqueried input. This prediction argument allows for the analysis of both the legality and the complexity of an extracted pebbling, and gives the final bound in terms of the CC of the underlying graph.

Theorem 1 (Amortized strong memory-hardness). *Let $q, t, m, n, \delta, K, N \in \mathbb{N}$. Let $\mathcal{A}$ be a q-query t-time adversary with cumulative memory complexity* $\mathrm{CMC}(\mathcal{A})$ *and $\mathcal{G}$ be a DAG. Let C^{H} be the iMHF based on $\mathcal{G}$ in the $(nN^{\delta}K, N)$-RO model as defined in Section 2.2. Then*

$$\mathbf{Adv}^{(q,m)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \frac{2q^2}{N} + 2^{-\gamma} \ ,$$

where $\gamma := \big(m \cdot \mathrm{CC}(\mathcal{G})[\log N - \log(qm)] - \mathrm{CMC}(\mathcal{A})\big)/t$.

Intuitively, γ captures the gap between the expected CMC of an adversary computing m instances following the canonical pebbling algorithm (where for each pebble, the adversary queries the hash to compute the next label) and the actual CMC of $\mathcal{A}$ averaged over t steps. Thus, γ becomes large as the adversarial CMC decreases beyond the minimum required for the construction. We give the proof in Appendix B.

In our extension, we have to account for adversaries that choose inputs in a hash-dependent way, and compute multiple instances of the construction. Note that if an illegal pebble placement occurs in the extracted pebbling, then a "bad" query in the transcript contains a correct response to some other "target" random oracle query, before the query is made. In the weaker memory hardness game, a predictor, given the index of this bad query as a hint, can recompute the response to the target query from the input.

Crucially, the adaptivity of picking inputs in the strong memory-hardness game means legality is *not* defined until $\mathcal{A}$ completes its run and the inputs become known. Thus in our reduction, we give an extended hint, to allow the

predictor to run $\mathcal{A}$ safely without querying the point for which it makes the prediction. The extension to the hint contains the position in the transcript, if there, of the query for which the adversary will predict the output.

We make a further extension in order to analyze the pebbling complexity, where again we reduce to a prediction game. This time, to run $\mathcal{A}$ safely, the predictor must detect for which instances $\mathcal{A}$'s queries are correct without necessarily knowing the input to the instance. This again requires an extension to the hint, that contains the location in the transcript of the starting queries for each of the instances the adversary needs to detect. These extensions allow us to give an upper bound on the advantage an adversary has in computing these graph-based iMHFs, showing that they are also *strongly* memory-hard.

4 Unrecoverability of Memory-Hard Functions

We now consider the multi-instance unrecoverability of graph-based iMHFs. Our result below establishes security bound in terms of the adversary's CMC (as opposed to query complexity). Its proof relies on adversarial upper bounds in computing an iMHF on m distinct hash-dependent inputs (which was established in the previous section).

Theorem 2 (UR security of C^{H}). *Let $P, q, n, m, c, t, \delta, \ell \in \mathbb{N}$, $\mathcal{P}_m$ be a m-sampler, $\mathsf{Gen} = [K]$, $\mathcal{G}$ be a DAG on n vertices, and C^{H} be the iMHF based on the graph $\mathcal{G}$ in the $(nN^{\delta}K, N)$-RO model as defined in Section 2.2. Then for any q-query, t-time, c-corruption adversary $\mathcal{A}$ in the* UR *game with respect to $\mathcal{P}_m, \mathsf{Gen}$ and C^{H}, there exists a P-query, c-corruption adversary $\mathcal{B}$ against* SA-GUESS *s.t.*

$$\mathbf{Adv}^{\mathrm{UR}}_{\mathcal{P}_m,\ell,\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + 2^{-\gamma} + \frac{(2q_{\mathcal{P}} + nm\ell + 4q)^2}{N} + \frac{m\ell}{K} \ ,$$

where $\gamma = [(P+1)\mathrm{CC}(\mathcal{G})[\log N - \log((q)m)] - (\mathrm{CMC}(\mathcal{A}) + \Delta)]/t$, and $\Delta = \mathrm{CMC}(\mathcal{P}_m) + mt \log N$.

P is chosen to minimize $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + 2^{-\gamma} + \frac{(nm\ell+4q)^2}{N}$ (see Appendix C).

We start with the multi-instance unrecoverability game and then move to a game that lazily samples the RO, and sets a bad flag and aborts on collisions in outputs. By aborting on collisions, we stop the adversary from choosing distinct inputs that collide early on in the computation, making it easier to amortize the computation. We then transition to a game that sets a bad flag and aborts if the adversary makes a query for an index > 1 that had already been made during the challenge generation phase, but where $\mathcal{A}$ has not computed the chain up to the node i. This ensures that an adversary cannot start the computation midway through and then try to recover a valid starting point. From here, we go through a series of game transitions that lead to the game no longer storing intermediate points generated during the challenge generation phase. Finally, we transition to a game where we abort if the adversary computes C^{H} on more than P distinct inputs, which we justify via a reduction to the $(q, P+1)$-sMH game defined in

Section 3. Having bounded the number of full computations the adversary can perform, we finally reduce to the SA-GUESS game, where the adversary is able to make P queries to the TEST oracle. By reducing to SA-GUESS, we only need to account for full computations on inputs the adversary chooses, as only the final H query in a full computation is forwarded to TEST. Note that in FT21, which considers the monolithic random oracle, every H query is forwarded to TEST.

Proof. We prove the theorem via a sequence of games as follows.

G_0: This game is defined to be the UR game with respect to $\mathcal{P}_m, \mathsf{Gen}$ and C^{H}. Thus, $\Pr[\mathrm{G}_0] = \mathbf{Adv}^{\mathrm{UR}}_{\mathcal{P}_m,\ell,\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{A})$.

G_1: This game lazily samples the random oracle. This game is identical to G_0: $\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_0]$.

G_2: This game sets a bad flag bad_1 whenever the output point sampled for any oracle $i < n$ has already been used as part of an input to the oracle indexed with any $j \in \mathbf{Ch}(i)$ or there is a collision in the hash outputs. (This event ensures that no two computations on distinct inputs collide – note that this includes computations from the challenge generation phase.) The game then aborts after bad_1 is set. Games G_1 and G_2 are identical until bad_1. As the outputs are sampled randomly, and the game (that is the adversary and the challenge generation phase) make a total of at most $q + mn$ queries to the oracles, we have $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \Pr[\mathsf{bad}_1] \leq \frac{(nm\ell+2q)^2}{N}$.

G_3: This game sets bad_2 when $\mathcal{A}$ makes a query containing a label generated during the challenge phase, without first completing the computation from the corresponding input, and then aborts after bad_2 is set. The check for computation from an input to a point within the computation is done using a procedure called FINDCHAIN. Given a label y and a point v, FINDCHAIN checks if there exists a computation chain from an initial input up to the point v and then either returns the input point or returns $\perp$. On input (y, v), FINDCHAIN runs as follows, it parses the input y into $v, \mathbf{lab}(\mathbf{Pa}_1(v)), \ldots, \mathbf{lab}(\mathbf{Pa}_d(v))$, for each $\mathbf{lab}(\mathbf{Pa}_i(v))$ it looks in the table for a query that returned $\mathbf{lab}(\mathbf{Pa}_i(v))$. Due to the bad events in the previous games for each $\mathbf{lab}(\mathbf{Pa}_i(v))$ there will be a at most one inverse y'. If no inverse exists then the function returns $\perp$, otherwise it recursively applies FINDCHAIN on $(y', \mathbf{Pa}_i(v))$. When FINDCHAIN is called on $(y, 1)$, i.e., the first hash in the computation, if it has an inverse in the table, FINDCHAIN has found the initial input to the construction and returns that input. Formally, for each password-salt pair (pw, sa), the queries made by the game to evaluate $\mathsf{C}^{\mathsf{H}}(pw, sa)$ are assigned to each node v a label l. If $\mathcal{A}$ makes a query containing the label l, without fully computing the $\mathsf{C}^{\mathsf{H}}(pw, sa)$ up to the corresponding node v, then bad_2 is set. We show in the challenge point prediction claim below that $\Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_2] \leq \Pr[\mathsf{bad}_2] \leq m\ell q/N$.

G_4: In this game, when the adversary makes a query for the final node, the game uses FINDCHAIN (described in G_3), to check if the query completes a full computation starting from a password-salt pair $(\mathbf{pw}[i], \mathbf{sa}[i, j])$. If it finds

such a computation, it responds with the corresponding challenge value, else it samples a fresh random value. The bad_2 flag ensures that $\mathcal{A}$ only labels points from the challenge generation correctly via a full computation from an input to that point. The bad_1 flag ensures $\mathcal{A}$ only computes in the forward direction. Together these flags ensure that the game will stay consistent with the challenge values $\mathbf{y}$, because any correct query $\mathcal{A}$ makes to label a sink node will be the result of a full chain, and so the game can respond with the right challenge point. Thus $\Pr[\mathrm{G}_4] = \Pr[\mathrm{G}_3]$.

G_5: In this game, we no longer store the intermediate points generated in the challenge phase, and if during the online phase $\mathcal{A}$ evaluates the construction on a correct password-salt pair the points are freshly sampled. Until a query from $\mathcal{A}$ resamples these intermediate points, the values are information theoretically hidden from the adversary. Thus: $\Pr[\mathrm{G}_5] = \Pr[\mathrm{G}_4]$.

G_7: This game sets a bad flag bad_3 whenever the adversary fully computes C^{H} on more than P unique inputs, and then aborts when bad_3 is set. Games G_7 and G_6 are identical until bad_3. We show, via a reduction to a strong $(q, P+1)$-sMH game, in the bounded chain completion claim below that $\Pr[\mathrm{G}_7] - \Pr[\mathrm{G}_6] \leq \Pr[\mathsf{bad}_3] \leq \mathbf{Adv}^{(q,P+1)\text{-sMH}}_{\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{B})$. We conclude the proof by showing in the G_7 to sa-Guess reduction claim below that the advantage of any adversary in G_7 is bounded by one in the sa-Guess.

Claim (Collisions). For parameters as above, we have

$$\Pr[\mathcal{A} \text{ sets } \mathsf{bad}_1 \text{ in } \mathrm{G}_2] \leq \frac{(nm\ell + 2q)^2}{N} \ .$$

The proof of this claim follows from a simple collision bound of any of the hash outputs colliding.

Claim (Challenge point prediction). For the parameters above, we have

$$\Pr[\mathsf{bad}_2 \text{ is set in } \mathrm{G}_3] \leq \frac{m\ell q}{N} \ .$$

The proof of this claim follows from a simple collision bound of any of the $m\ell$ challenge points colliding with any of the queries of $\mathcal{A}$.

Claim (Bounded chain completion). Setting the parameters as above, let $\mathrm{CMC}(\mathcal{B}) = \mathrm{CMC}(\mathcal{A}) + mt \log N$, then:

$$\Pr[\mathcal{A} \text{ sets } \mathsf{bad}_3 \text{ in } \mathrm{G}_7] \leq \mathbf{Adv}^{(q,P+1)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{B}) \leq 2^{-\gamma} + \frac{2q^2}{N} \ ,$$

where $\gamma := ((P+1) \cdot \mathrm{CC}(\mathcal{G})(\log N - \log(qm) - \mathrm{CMC}(\mathcal{B}))/t$.

Proof. Given an adversary $\mathcal{A}$ as defined in the theorem statement, we construct a $(q + q_{\mathcal{P}})$-query adversary $\mathcal{B}$ against $(q + q_{\mathcal{P}}, P+1)$-sMH, where $\mathrm{CMC}(\mathcal{B}) = \mathrm{CMC}(\mathcal{A}) + q_{\mathcal{P}} + mt \log(N)$. Algorithm $\mathcal{B}$, with access to H, samples a random

password vector and leakage z from $\mathcal{P}^{\mathsf{H}}_m$, samples $m\ell$ salt vectors from Gen and generates a random challenge output vector $\mathbf{y}$ which it sends to $\mathcal{A}$, along with the salt vectors and leakage z. $\mathcal{B}$ then runs $\mathcal{A}$ and forwards each of $\mathcal{A}$'s queries to its oracle.

When $\mathcal{A}$ makes a query with the index of the source node, i.e., a query of the form $(1, l)$, algorithm $\mathcal{B}$ appends the input l to its special output register $\sigma_{\mathcal{O}}$. When $\mathcal{A}$ makes a query to label the sink node n, i.e., a query of the form (n, l), $\mathcal{B}$ appends the response $\mathsf{H}(n, l)$ to the register $\sigma_{\mathcal{O}}$. For any COR query $\mathcal{A}$ makes, $\mathcal{B}$ responds with the corresponding password. When $\mathcal{A}$ terminates, $\mathcal{B}$ returns $\sigma_{\mathcal{O}}$, which contains at most q values.

To handle $\mathcal{A}$'s corruption queries, $\mathcal{B}$ stores the password vector $\mathbf{pw}$. This increases $\mathrm{CMC}(\mathcal{B})$ by $mt\log(N)$. As algorithm $\mathcal{B}$ recognizes each input or output query as it receives it, and appends stored values to $\sigma_{\mathcal{O}}$ immediately, it runs $\mathcal{A}$ with no extra CMC cost. If $\mathcal{A}$ triggers bad_4, then there exists a set of $(P+1)$ correct input-output pairs in $\sigma_{\mathcal{O}}$, and $\mathcal{B}$ wins the $(q+q_{\mathcal{P}}, P+1)$-sMH game. □

Claim (G_6 *to* SA-GUESS *reduction).* Let $P, q, n, m, c \in \mathbb{N}$, $\mathcal{G}$ a graph on n vertices with maximum in-degree δ, and C^{H} the iMHF based on $\mathcal{G}$ in the $(nN^{\delta}K, N)$-RO model defined in Section 2.2. Then, for any m-sampler $\mathcal{P}_m$, any unpredictable salt generator Gen, and any q-query c-corruption adversary $\mathcal{A}$ against G_6, there exists a P-query, c-corruption adversary $\mathcal{B}$ against SA-GUESS s.t.

$$\mathbf{Adv}^{\mathrm{G}_6}_{\mathcal{P}_m,\ell,\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) \ .$$

Proof. Given an adversary $\mathcal{A}$ as in the theorem statement in G_6, we construct an adversary $\mathcal{B}$ against SA-GUESS as follows. We assume, without loss of generality, that $\mathcal{A}$ only returns a value x if it has computed $\mathsf{C}^{\mathsf{H}}(x)$. [5] $\mathcal{B}$ receives the salt vector $\mathbf{sa}$, leakage z, and generates a challenge vector $\mathbf{y}$ following the rules of G_6. Algorithm $\mathcal{B}$ runs $\mathcal{A}$ on $\mathbf{y}$ and z and for all $\mathcal{A}$'s queries with node indices from 1 to $n-1$, $\mathcal{B}$ answers by lazily sampling the random oracle, if any of $\mathcal{B}$'s answers lead to a collision between computations with distinct start points, then $\mathcal{B}$ aborts according to the rules corresponding to bad_1 of G_2. For any queries that $\mathcal{A}$ makes to the oracle for the final node, i.e., of the form $(n, \ell_1, \ldots, \ell_\delta)$, algorithm $\mathcal{B}$ checks if past queries form a complete computation of the construction on some input (x, sa) using FINDCHAIN (described in G_3). If it does, $\mathcal{B}$ queries (x, sa) to TEST. If TEST returns true, $\mathcal{B}$ responds to $\mathcal{A}$'s query with the value $y \in \mathbf{y}$ corresponding to (x, sa), else it lazily samples the random oracle output. As we have a single sink, any query to the sink node must correlate to a full computation, note that if there were multiple sink nodes then a query to TEST must be made any time a sink node is reached, regardless of if the other sinks had been reached yet – i.e.

[5] For any $\mathcal{A}$ that returns x without computing $\mathsf{C}^{\mathsf{H}}(x)$ we can create an $\mathcal{A}'$ which runs $\mathcal{A}$ and computes $\mathsf{C}^{\mathsf{H}}(x)$ before returning x, and where $\mathrm{CMC}(\mathcal{A}') = \mathrm{CMC}(\mathcal{A}) + m' \cdot \mathrm{sCMC}_{q,1}(\mathsf{C}^{\mathsf{H}})$. We assume, without loss of generality, that m' is less than the number of guesses made by $\mathcal{A}$ - as otherwise it would be impossible for $\mathcal{A}$ to ever win, no matter how low entropy the password distribution, therefore $\mathrm{CMC}(\mathcal{A}') = \mathcal{O}(\mathrm{CMC}(\mathcal{A}))$, where m' is the number of passwords recovered.

regardless of if the whole computation had been made. $\mathcal{A}$ could have computed the function for at most P unique start points, otherwise bad_2 would have been triggered and the game would have aborted. As $\mathcal{B}$ will have queried each of the P start points (on which $\mathcal{A}$ computed C^{H}) to its TEST oracle, and $\mathcal{A}$ only returns values on which it has performed full computations, then any (x', sa') that $\mathcal{A}$ returns will have been queried to TEST by $\mathcal{B}$. Note that throughout the proof, even with a single challenge, if the adversary correctly "guesses" the password but doesn't fully complete the computation, we do not set a bad flag. As $\mathcal{B}$ will have returned y if TEST returns true, then whenever $\mathcal{A}$ wins, $\mathcal{B}$ will also win. □

Combining the game transitions above gives

$$\begin{aligned}\Pr[\mathrm{G}_0] &= \Pr[\mathrm{G}_6] + \sum_{i=0}^{5}(\Pr[\mathrm{G}_i] - \Pr[\mathrm{G}_{i+1}]) \\ &\leq \mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(P,c) + \frac{2q^2}{N} + 2^{-\gamma} + \frac{m\ell q}{N} + \frac{(nm\ell+2q)^2}{N} \\ &\leq \mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(P,c) + 2^{-\gamma} + \frac{(nm\ell+4q)^2}{N} \ ,\end{aligned}$$

where the last inequality is as a result of

$$(nm\ell+4q)^2 \geq 2q^2 + m\ell q + (nm\ell+2q)^2 \ ,$$

and that $\gamma := ((P+1)\cdot \mathrm{CC}(\mathcal{G})[\log N - \log(qm)] - (\mathrm{CMC}(\mathcal{A}) + \Delta))/t$, and besides $\Delta = \mathrm{CMC}(\mathcal{P}_m) + mt\log N$. □

One-wayness. Password-storage security can also be analyzed with respect to a standard one-wayness security notion where the adversary's goal is to recover *any* input as long as it produces the same construction output as that of the actual password. Moving between unrecoverability and one-wayness can be achieved in a straightforward manner up to the collision resistance of the construction, which can be shown to be $\mathcal{O}(q^2)/N$ for any q-query adversary, i.e., we get:

$$\mathbf{Adv}^{\mathrm{OW}}_{\mathcal{P}_m,\ell,\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{UR}}_{\mathcal{P}_m,\ell,\mathsf{Gen},\mathsf{C}^{\mathsf{H}}}(\mathcal{B}) + \frac{\mathcal{O}(q^2)}{N} \ .$$

This bound is sufficient for password-hashing purposes since passwords are typically low entropy and thus their unguessability under the UR bound dominates the term due to hash collisions. This means that even if the format of the passwords is not checked and any value is accepted as an input by a password-based system, security remains intact.

References

1. Password hashing competition, `https://www.password-hashing.net/`, `www.password-hashing.net`

2. Alwen, J., Chen, B., Pietrzak, K., Reyzin, L., Tessaro, S.: Scrypt is maximally memory-hard. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 33–62. Springer, Cham (Apr / May 2017). https://doi.org/10.1007/978-3-319-56617-7_2
3. Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: Servedio, R.A., Rubinfeld, R. (eds.) 47th ACM STOC. pp. 595–603. ACM Press (Jun 2015). https://doi.org/10.1145/2746539.2746622
4. Bellare, M., Ristenpart, T., Tessaro, S.: Multi-instance security and its application to password-based cryptography. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 312–329. Springer, Berlin, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_19
5. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) ACM CCS 93. pp. 62–73. ACM Press (Nov 1993). https://doi.org/10.1145/168588.168596
6. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: IEEE European Symposium on Security and Privacy (EuroS&P). pp. 292–302 (2016). https://doi.org/10.1109/EuroSP.2016.31
7. Boneh, D., Corrigan-Gibbs, H., Schechter, S.E.: Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 220–248. Springer, Berlin, Heidelberg (Dec 2016). https://doi.org/10.1007/978-3-662-53887-6_8
8. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Berlin, Heidelberg (Oct 2006). https://doi.org/10.1007/11894063_16
9. Farshim, P., Tessaro, S.: Password hashing and preprocessing. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part II. LNCS, vol. 12697, pp. 64–91. Springer, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77886-6_3
10. Forler, C., Lucks, S., Wenzel, J.: Catena: A memory-consuming password scrambler. Cryptology ePrint Archive, Report 2013/525 (2013), https://eprint.iacr.org/2013/525
11. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 617–630. Springer, Berlin, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4_36
12. Percival, C., Josefsson, S.: The scrypt Password-Based Key Derivation Function. RFC 7914, IETF (2016). https://doi.org/10.17487/RFC7914, https://www.rfc-editor.org/info/rfc7914
13. PKCS #5: Password-based cryptography specification. RSA Data Security, Inc. (Sep 2000), version 2.0
14. Provos, N., Mazières, D.: A future-adaptable password scheme. In: Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA. pp. 81–91. USENIX (1999), http://www.usenix.org/events/usenix99/provos.html

A Amortizing Scrypt over Hash-Dependent Inputs

Here, we give a separation between the hash-independent and hash-dependent notions of CMC. We first give a definition of what it means for a construction

to have a high CMC in our stronger setting where inputs are chosen in a hash dependent manner. To do this we use the sMH game.

CMC *of a construction.* We say $\mathcal{A}$ ϵ-computes m instances of C^{H} if $\mathbf{Adv}^{(m,m)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \geq \epsilon$. Given a construction C^{H}, we define its *strong* (q,ϵ)-sCMC as the CMC of the "best" q-query algorithm that computes C^{H} on some (typically the easiest) input with probability at least ϵ. That is,

$$\text{sCMC}_{q,\epsilon}(\mathsf{C}^{\mathsf{H}}) := \min_{\mathcal{A} \in S[q,\epsilon,\mathsf{C}^{\mathsf{H}}]} \{\text{CMC}(\mathcal{A})\} \ ,$$

where $S[q,\epsilon,\mathsf{C}^{\mathsf{H}}]$ is the set of all q-query algorithms $\mathcal{A}$ in pROM with advantage at least ϵ in the $(1,1)$-sMH game against C^{H}. Similarly, for $m \in \mathbb{N}$, we define the (m,q,ϵ)-sCMC of C^{H} as the CMC of the best q-query algorithm that computes C^{H} on at most m (possibly hash-dependent) instances with probability at least ϵ, where the effort is scaled by the number of instances computed. In other words,

$$\text{sCMC}_{m,q,\epsilon}(\mathsf{C}^{\mathsf{H}}) := \min_{\bar{m} \in [m]} \min_{\mathcal{A} \in S[q,\epsilon,\bar{m},\mathsf{C}^{\mathsf{H}}]} \{\text{CMC}(\mathcal{A})/\bar{m}\} \ ,$$

where $S[q,\epsilon,\bar{m},\mathsf{C}^{\mathsf{H}}]$ is the set of all q-query algorithms $\mathcal{A}$ in pROM that have advantage at least ϵ in the $(\bar{m},\bar{m})$-sMH game against C^{H}. We say that a construction C is (m,q,ϵ,μ)-memory-hard if its (m,q,ϵ)-CMC is at least μ. Note that when $m = 1$, we recover the single-instance notion defined above. Intuitively, these definitions describe the average cumulative memory cost *per instance*, for the best adversary succeeding with probability at least ϵ. Measuring the cost per instance allows us to capture the *non-amortizability* of a construction. Informally, any savings in CMC will be captured by this metric, even if they are the result of sharing memory across computations for (possibly many) different instances.

Separation. In [2], the authors prove that ROMix, the main component of Scrypt, is maximally memory-hard for computing a single instance in the hash-independent setting, and claim that ROMix is maximally memory-hard when computing multiple instances. We show that this is not the case in the hash-dependent setting by giving an algorithm that computes n instances with an overall (hash-dependent) sCMC of $\mathcal{O}(n^2 \log N)$, as opposed to an overall (hash-independent) CMC of $\Omega(n^3 \log N)$. This in turn translates to an amortized sCMC of $\mathcal{O}(n \log N)$ per instance, as opposed to an amortized CMC of $\Omega(n^2 \log N)$ per instance.

Note that the sCMC definition extends adversarial capabilities in two ways. First, is the hash-dependent inputs, and second is the unordered output set. Importantly, the attack provided in this section only leverages the hash-dependent inputs. That is, it acts as a separation for the hash-dependent and hash-independent security models, irrespective of any other strengthening provided by the sMH game.

Scrypt. As with [2], we focus on the ROMix construction. ROMix is defined in terms of an integer n and a hash function $\mathsf{H} \in \mathrm{Fun}(2N, N)$, which we will model as a random oracle, and takes as input a value X. ROMix computes the values $X_0, \ldots, X_{n-1}, S_0, \ldots, S_n$, as defined in Fig. 3, and outputs S_n.

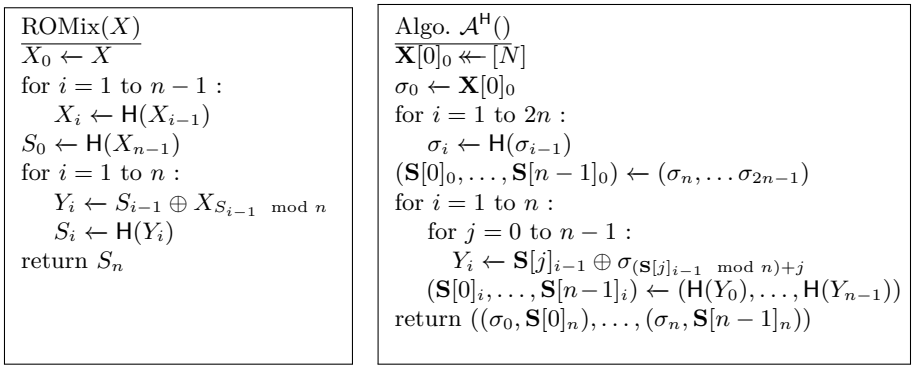

Fig. 3: left: The ROMix construction, right: An algorithm that computes ROMix on n inputs, with a sCMC that is n times lower than the sCMC required for a single instance.

ROMix can be seen to be comprising two chains: the X chain containing $X_0, \ldots, X_{n-1}$ and the S chain containing $S_0, \ldots, S_n$. A canonical computation of ROMix for one input requires computing the full X chain of length n, and retaining it in memory for the n steps of the S chain computation, resulting in a CMC of $\mathcal{O}(n^2 \log N)$ as each stored value requires $\log N$ bits. Alwen et al. [2] essentially show that this is the best any computation strategy could hope for.

More precisely, in [2], the authors show than any algorithm that attempts to compute Scrypt on a vector of n inputs chosen independently of the hash function will (with overwhelming probability) require an overall CMC of $\Omega(n^3 \log N)$ and hence an amortized CMC of $\Omega(n^2 \log N)$. The authors give the following intuition. An algorithm storing a subset of the labels from the initial X values, say of size p, can expect each X value it needs to recall (for the computation of the S values in the second phase) to be on average $n/(3p)$ iterations from the nearest stored label. The required CMC for this strategy is $(p \cdot n \cdot \log N)/2p = (n \log N)/2$. As there are n labels to compute, the algorithm achieves the bound above. However, we show that to compute Scrypt on the inputs $\mathbf{X} = (X, \mathsf{H}(X), \mathsf{H}^2(X), \ldots, \mathsf{H}^{n-1}(X))$ requires an overall sCMC of $\Omega(n^2 \log N)$ and hence an amortized sCMC of just $\mathcal{O}(n \log N)$ per instance. Intuitively, the algorithm we propose chooses inputs which lead to overlaps in the hash chains that are computed in ROMix. It can do this by picking points along a hash chain it generates in the first for loop, i.e. the X chain. These overlaps reduce the cumulative memory cost by sharing the cost of retaining the

chain between all instances. We then further generalize this to an algorithm that computes ROMix on k distinct inputs, with an amortized sCMC of $o(n \log N)$.

Notation. We use $\mathbf{X}[i]_j$ and $\mathbf{S}[i]_j$ to respectively represent the values X_j and S_j computed for the input $\mathbf{X}[i]$. We consider cases where the algorithm is computing ROMix on more than one input, specifically on k inputs.

Warm-up: $k = n$. We start by considering the case where $k = n$, i.e. the algorithm is computing ROMix on n distinct inputs. We use this as a warm-up case as it simplifies the calculations and makes clear how the sCMC is reduced.

Lemma 1. *Let $n, N \in \mathbb{N}$. Then in the (N^2, N)-pRO model, there exists an algorithm that computes ROMix on n distinct inputs with a total* sCMC *of $\mathcal{O}(n^2 \log N)$, thus giving an amortized* sCMC *of $\mathcal{O}(n \log N)$.*

Proof. Our algorithm runs in the parallel random-oracle model, we give pseudo-code for it in Fig. 4 in the style of an algorithm in the pROM from Fig. 1, we give slightly less formal pseudo-code for it in Fig. 3 and describe it here.

The algorithm computes $\mathbf{X}[1]_1$ to $\mathbf{X}[1]_{2n}$ from the first input, as the other $n-1$ inputs are generated by hashing the first input, both they and their corresponding $\mathbf{X}[i]_1$ to $\mathbf{X}[i]_n$ are contained within $\mathbf{X}[1]_1, \ldots, \mathbf{X}[1]_{2n}$. For computing $\mathbf{S}[i]_1, \ldots, \mathbf{S}[i]_n$ the algorithm follows the canonical approach.

More formally, for rounds $i = 1$ to $2n$, algorithm $\mathcal{A}$ outputs the pair $(\sigma_i, \mathbf{q}_i) = (\sigma_{i-1} || \mathsf{H}(X_{i-1}), X_i)$. After these $2n$ rounds, σ_{2n} contains $X, \mathsf{H}(X), \ldots, \mathsf{H}^{2n}(X)$. Note that $|\sigma_{2n}| = 2n \log N$, and across these $2n$ rounds sCMC$(\mathcal{A}) = (2n)(2n-1) \log N/2 \leq 4n^2 \log N$, and that this contains the corresponding $X_0, \ldots X_{n-1}$ values for $\mathbf{X} = [X, \mathsf{H}(X), \ldots, \mathsf{H}^{n-1}(X)]$. Note that there is a lot of overlap, e.g. $\mathbf{X}[0]_n = \mathbf{X}[1]_{n-1} = \mathbf{X}[2]_{n-2} = \ldots = \mathbf{X}[n-1]_1$

For each round $i = 2n+1$ to $3n$, $\mathcal{A}$ computes S_i for each input X in parallel. Specifically, at the end of each round i, $\mathcal{A}$ outputs the pair $(\sigma_i, \mathbf{q}_i) = (\sigma_{2n}, \mathbf{S}[0]_{i-1} \oplus Y_{\mathbf{S}[0]_{i-1} \mod n}, \ldots, \mathbf{S}[n-1]_{i-1} \oplus Y_{\mathbf{S}[n-1]_{i-1} \mod n})$. For each of these rounds the size of $(\sigma_i, \mathbf{q}_i)$ is $2n \log N + n \log N = 3n \log N$, giving sCMC$(\mathcal{A})$ for rounds $i = 2n+1$ to $3n$ as $3n^2 \log N$.

At the start of round $3n$, $\mathcal{A}$ receives the values $\mathbf{S}[0]_n, \ldots, \mathbf{S}[n-1]_n$, which it outputs. Therefore the overall sCMC$(\mathcal{A})$ is $7n^2 \log N$, which when amortized over the n values being computed simultaneously, gives an amortized sCMC$(\mathcal{A})$ of $7n \log N$, far below the original hash-independent bound. □

General case: any (non-constant) $k = \mathcal{O}(n)$. We now show how the above lemma generalizes to computing ROMix on any (non-constant) $k = \mathcal{O}(n)$ instances, rather than $k = n$ only. Specifically, we show that for such k, we can get an amortized sCMC of $o(n^2 \log N)$ which is explicitly smaller than $\Omega(n^2 \log n)$.

Theorem 3. *Let $N, n, k \in \mathbb{N}$, where $k = \omega(1)$ and $k = \mathcal{O}(n)$. Then in the (N^2, N)-pRO model, there exists an algorithm that computes ROMix on k distinct inputs whose amortized* sCMC *is $o(n^2 \log N)$.*

Algo. $\mathcal{A}_*^{\mathsf{H}}(z)$
$\sigma_0 \leftarrow \emptyset$
for $i = 1$ to $3n$:
  $(\mathbf{q}_i, st) \leftarrow \mathcal{A}(i, z, \sigma_{i-1})$
  $\mathbf{y} \leftarrow \mathsf{H}(\mathbf{q}_i)$
  $(\sigma_i, \sigma_{\mathcal{O}}) \leftarrow \mathcal{A}(\mathbf{y}; st)$
return $(\sigma_{\mathcal{O}})$

Algo. $\mathcal{A}_b(i, \mathbf{y}, st)$
if $i \in [1, 2n]$:
  return $((st, \mathbf{y}), \sigma_{\mathcal{O}})$
else if $i \in [2n+1, 3n-1]$:
  for $j = 1$ to n :
    $st[n + j - 1] \leftarrow \mathbf{y}[j-1]$
  return $(st, \emptyset)$
else if $i = 3n$:
  $\sigma_{\mathcal{O}} \leftarrow \emptyset$
  for $j = 1$ to n :
    $\sigma_{\mathcal{O}}[j-1] \leftarrow \mathbf{y}[j-1]$
  return $(\emptyset, \sigma_{\mathcal{O}})$

Algo. $\mathcal{A}_a(i, z, \sigma_{i-1})$
if $i = 1$:
  return (z, σ_{i-1})
else if $i \in [2, 2n]$:
  return $(\sigma_{i-1}[i], \sigma_{i-1})$
else if $i = 2n + 1$:
  $(\mathbf{S}[0]_0, \ldots \mathbf{S}[n-1]_0) \leftarrow (\sigma_{i-1}[n], \ldots \sigma_{i-1}[2n-1])$
  for $j = 0$ to $n - 1$:
    $Y[j]_1 \leftarrow \mathbf{S}[j]_0 \oplus \sigma_{i-1}[(\mathbf{S}[j]_0 \mod n) + j]$
  return (Y_1, σ_{i-1})
else if $i \in [2n + 2, 3n]$:
  $r \leftarrow i - 2n - 1$
  $(\mathbf{S}[0]_r, \ldots \mathbf{S}[n-1]_r) \leftarrow (\sigma_{i-1}[2n], \ldots \sigma_{i-1}[3n-1])$
  for $j = 0$ to $n - 1$:
    $Y[j]_r \leftarrow \mathbf{S}[j]_r \oplus \sigma_{i-1}[(\mathbf{S}[j]_r \mod n) + j]$
  return $((Y[0]_r, \ldots, Y[n]_r), \sigma_{i-1})$

Fig. 4: A parallel algorithm written in the style of a pROM algorithm that computes ROMix on n inputs, with an amortized sCMC that is n times lower than the sCMC required for a single instance.

Proof. Intuitively, the algorithm follows that of Lemma 1, but only computes $\mathbf{X}[1]_1, \ldots, \mathbf{X}[1]_{n+k}$.

More formally, for rounds $i = 1$ to $n + k$, algorithm $\mathcal{A}$ outputs the pair $(\sigma_i, \mathbf{q}_i) = (\sigma_{i-1} || \mathsf{H}(X_{i-1}), X_i)$. After these $n+k$ rounds, σ_{n+k} contains the chain $X, \mathsf{H}(X), \ldots, \mathsf{H}^{n+k}(X)$. Note that $|\sigma_{n+k}| = (n+k)\log N$, and across these $n+k$ rounds $\text{sCMC}(\mathcal{A}) = (n+k)(n+k-1)\log N/2 \leq (n+k)^2 \log N$, and that this contains the corresponding $X_0, \ldots X_{n-1}$ values for $\mathbf{X} = [X, \mathsf{H}(X), \ldots, \mathsf{H}^{k-1}(X)]$. Note that there is some overlap, e.g. $\mathbf{X}[0]_n = \mathbf{X}[1]_{n-1} = \mathbf{X}[2]_{n-2} = \ldots$.

For each round $i = n + k + 1$ to $n + 2k$, algorithm $\mathcal{A}$ computes S_i for each input X in parallel. Specifically at the end of each round i, $\mathcal{A}$ outputs the pair $(\sigma_i, \mathbf{q}_i) = (\sigma_{n+k}, \mathbf{S}[0]_{i-1} \oplus Y_{\mathbf{S}[0]_{i-1} \mod n}, \ldots, \mathbf{S}[n-1]_{i-1} \oplus Y_{\mathbf{S}[n-1]_{i-1} \mod n})$. For each of these rounds the size of $(\sigma_i, \mathbf{q}_i)$ is $(n+k)\log N + k \log N = (n+2k)\log N$, giving $\text{sCMC}(\mathcal{A})$ for rounds $i = n + k + 1$ to $n + 2k$ as $k(n + 2k)\log N$.

At the start of round $n+2k$, $\mathcal{A}$ receives the values $\mathbf{S}[0]_n, \ldots, \mathbf{S}[k-1]_n$, which it outputs. Therefore, the overall $\text{sCMC}(\mathcal{A})$ is $k(n+2k)\log N + (n+k)^2 \log N = (n^2 + 3nk + 3k^2)\log N$, which when amortized over the k values being computed simultaneously, gives an amortized $\text{sCMC}(\mathcal{A})$ of $(n^2/k + 3n + 3k)\log N$. When $k = \omega(1)$ and $k = \mathcal{O}(n)$, this gives $\text{sCMC}(\mathcal{A}) = o((n^2 + n + k)\log N) = o(n^2 \log N)$.

Note that this shows that even for small values of k, as long as $k = \omega(1)$, then there is an amortizing algorithm that beats the CMC bound of $\Omega(n^2 \log n)$. □

B Proof of Theorem 1

We first recall two lemmas from [3] used in our proof. The first lemma concerns the additive property for the cumulative complexity of two graphs [3, Lemma 3].

Lemma 2 (Additive property of $\mathrm{CC}(\mathcal{G})$ [3]). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two node-disjoint DAGs and let $\mathcal{G}$ be their disjoint union then,* $\mathrm{CC}(\mathcal{G}) = \mathrm{CC}(\mathcal{G}_1) + \mathrm{CC}(\mathcal{G}_2)$.

Consequently, for any positive integer m and any DAG $\mathcal{G}$, the minimum cost to pebble m instances of $\mathcal{G}$ is given by $\mathrm{CC}(\mathcal{G}^{\times m}) = m \cdot \mathrm{CC}(\mathcal{G})$.

The second lemma concerns the predictor bound [3, Lemma 1], stating that an algorithm, given a hint, can successfully predict a response from a random oracle with probability bound by the size of the hint space divided by the size of range of the random oracle.

Lemma 3 (Predictor bound [3]). *Let $B = b_1, ..., b_u$ be a sequence of random bits. Let $\mathcal{P}'$ be a randomized procedure which gets a hint $h \in \mathrm{H}$, and can adaptively query any of the bits of B by submitting an index i and receiving b_i. At the end of the execution, $\mathcal{P}'$ outputs a subset $S \subseteq [u]$ of $|S| = k$ indices which were not queried, along with guesses for all $\{b_i | i \in S\}$. Then the probability (over the choice of B and randomness of $\mathcal{P}'$) that there exists some $h \in \mathrm{H}$ for which $\mathcal{P}'(h)$ outputs* all correct guesses *is at most $|\mathrm{H}|/2^k$.*

We now recall the statement of Theorem 1 and present the proof.

Theorem 1 (Amortized strong memory-hardness). *Let $q, t, m, n, \delta, K, N \in \mathbb{N}$. Let $\mathcal{A}$ be a q-query t-time adversary with cumulative memory complexity $\mathrm{CMC}(\mathcal{A})$ and $\mathcal{G}$ be a DAG. Let C^{H} be the iMHF based on $\mathcal{G}$ in the $(nN^{\delta}K, N)$-RO model as defined in Section 2.2. Then*

$$\mathbf{Adv}^{(q,m)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \frac{2q^2}{N} + 2^{-\gamma} \ ,$$

where $\gamma := \big(m \cdot \mathrm{CC}(\mathcal{G})[\log N - \log(qm)] - \mathrm{CMC}(\mathcal{A})\big)/t$.

In [3] the proof in the single-instance hash-independent setting proceeds via two claims. The first claims that a pebbling P, extracted from a transcript via the ex-post-facto pebbling extraction algorithm, is legal with high probability. The second claims that, with high probability, the total number of pebbles in the extracted P is upper bounded by an essentially linear function of $\mathrm{CMC}(\mathcal{A})$. Both claims rely on a certain "prediction argument", which, roughly speaking, states that a transcript giving rise to an illegal pebbling can be used to predict the output of a random oracle on a point without querying it. For an overview of how we adapt the techniques for our sMH game, see the main body in Section 3. Note that the following argument is independent of whether the construction is evaluated with a salt or not. Any salt used in evaluating the construction (and thus defining the labeling of the graph) is chosen by the adversary. Therefore, without loss of generality, we can consider the input to be a salt-input pair.

Proof. Fix all variables and graph as in the statement of the lemma. Let $\mathcal{A}$ be a q-query adversary in the (q, m)-sMH game with respect to C^{H}.

Transcript. An attack transcript T is a tuple $(\mathbf{q}_1, \ldots, \mathbf{q}_t, \sigma_O)$, where each $\mathbf{q}_i$ is itself a round of queries $(q_{i,1}, q_{i,2}, \ldots)$ consisting of $q_{i,j}$ either of the form $(x_{i,j}, y_{i,j})$, where $x_{i,j}$ is a hash input and $y_{i,j}$ is a hash output, or of the form $((v_{i,j}, \ell_{i,j}, out), \perp)$, where $v_{i,j}$ is a node, and $\ell_{i,j}$ is a labeling function output. Here, $\sigma_O = (\mathbf{x}, \mathbf{y})$ is the special output register that corresponds to the final output of the adversary.

Given a transcript of $\mathcal{A}$ we adapt the ex-post-facto pebbling extraction algorithm of [3] to extract pebbling P of the graph $\mathcal{G}^{\times m}$ *even in a hash-dependent setting.*

Overview. We start with a high-level overview. Before extracting a pebbling, we first determine, from the transcript, a vector $\mathbf{x}^*$ of m inputs for which $\mathcal{A}$ correctly computed the output of the construction. As we consider adversaries which may output many more values than they correctly compute, we first extract the subset of inputs which are relevant to the pebbling. For each $k \in [m]$, the input $\mathbf{x}^*[k]$ will define a labeling of the graph $\mathcal{G}$, and using these m computations, we extract a pebbling of the tensor graph $\mathcal{G}^{\times m}$. A pebbling $P = (P_0, P_1, \ldots, P_t)$ is extracted by a two-step procedure in which we process each $\mathbf{q}_i$ to calculate P_i by first combining all nodes that are *correctly labeled* by queries in $\mathbf{q}_i$ with the previous set P_{i-1}. Importantly, the node to be added to the pebbling set needs to be calculated from the index-input pair for which the label is correct. That is, if a node $v \in \mathcal{G}$, is correctly labeled with respect to some input $\mathbf{x}^*[k]$, the node (v, k) will be added to the set P_i. Next, we only keep *necessary nodes* in P_i—these are nodes whose labels are used in a later step and not recomputed in the meantime—and remove all others. We start by defining a (hash) function corresponding to a transcript and node-labels computed with respect to it, before formalizing these notions.

Pre-label of a node. Given $(\mathbf{q}_1, \ldots, \mathbf{q}_t)$ we define a function H where $H(x_{ij}) := y_{ij}$ if $(x_{ij}, y_{ij}) \in \mathbf{q}_i$ for some i, and otherwise $H(x) :=\perp$. We also set $H(\perp) :=\perp$.[6] As we are dealing with different labelings of the same graph, we need to extend the notation for labels and pre-labels. For each $k \in [m]$ let $\mathbf{lab}_k$ and $\mathbf{pre\text{-}lab}_k$ be the labeling and pre-labeling functions as defined in Section 2.2 with respect to H and $l := \mathbf{x}^*[k]$ (recovered from the transcript). Recall that a pre-label $\mathbf{pre\text{-}lab}_k(v)$ for some node v takes the form $(v, \mathbf{lab}_k(\mathbf{Pa}_1(v)), \ldots, \mathbf{lab}_k(\mathbf{Pa}_\delta(v)))$, where $\mathbf{Pa}$, as before, is the parent function. In what follows, all labeling and pre-labeling are performed with respect to the H and $\mathbf{x}^*$ extracted from the transcript. We make H implicit, but specify k.

Correct call. Intuitively, a correct call for a node in $\mathcal{G}$ is the pre-label of that node. Formally, a query (x, y) in the transcript is called correct (for some input $\mathbf{x}^*[k]$) if there exists some non-sink node v in the graph $\mathcal{G}$ s.t. $x = \mathbf{pre\text{-}lab}_k(v)$,

[6] We assume the transcript satisfies the unique-output-value property expected from a function.

or there exists a sink node v in the graph s.t. $x = (v, \mathbf{lab}_k(v), out)$. We call v the output-node corresponding to (x, y) and any parents of v in $\mathbf{Pa}(v)$ an input-node for (x, y). For the description of the procedure in pseudo-code, see Fig. 5.

Necessary node. Intuitively, a (pebbled) node is called unnecessary if the pebble on it is not used in a subsequent round to pebble a child. Formally, for any $k \in [m]$, and any node v in graph $\mathcal{G}$, the node $(v, k) \in P_i$ is called necessary in a round i if there exists a batch of queries $\mathbf{q}_j$ in a later round $j > i$ that contains a correct oracle call with respect to input $\mathbf{x}^*[k]$, where node v in (graph $\mathcal{G}$) is an *input-node*, but there is no batch $\mathbf{q}_p$ with $i < p < j$ that contains a correct *oracle call* for v. Note that we are translating from a node in the graph $\mathcal{G}^{\times m}$ to a node in $\mathcal{G}$, so correctness is defined by the $k-$th input in $\mathbf{x}^*$. A description of the procedure is given in Fig. 5.

Extraction. We now define the ex-post-facto pebbling-extraction algorithm. Given a transcript T, the extraction algorithm sets $P_0 := \emptyset$, and defines $P_1, \ldots, P_t$ by applying the following three rules to the each batch $\mathbf{q}_i$ in the order that they appear in the transcript.

1. Add all nodes in P_{i-1} to P_i.
2. For each query $x_{i,j}$ in the current batch, if $x_{i,j}$ is a correct call for some node v and input $\mathbf{x}^*[k]$, add (v, k) to P_i.[7]
3. For each (v, k) in P_i that is not necessary, remove it from P_i

Note that we say a node is *placed* in P (or pebbled) at some step i if it is added to P_i in step 2 above. This allows us to distinguish between the nodes inherited from a previous step and ones added due to a correct oracle call. See Fig. 5 for the pseudo-code describing the ex-post-facto pebbling extraction. With the pebbling extraction defined, we can analyze a pebbling extracted from a hash-dependent adversary. We now prove two claims. To prove the first (legality), we make a similar prediction argument to that made in [3], adapting the predictor to be suitable for hash-dependent adversaries by extending the predictor's hint, which replaces a q/N term by q^2/N. For the second (complexity), we again adapt the predictor's hint, this time to account for complications introduced by an adversary computing multiple instances of the construction. In particular, the predictor must know for which instance oracle calls are correct, and we pass this information for a certain set of queries, which replaces a $\log(q)$ term with $\log(mq)$ in the final bound.

Claim (Pebbling legality). For a fixed q-query pROM adversary $\mathcal{A}$, with fixed random coins and a fixed H, let T be a winning transcript corresponding to $\mathcal{A}$ against (q, m)-sMH. Then the pebbling P extracted from T is legal with probability at least $1 - q^2/N$, over the choice of H and the coins of $\mathcal{A}$.

An analogous result was proven in [3] for a pebbling of $\mathcal{G}$ extracted from a hash-independent adversary computing a single instance. In such a setting,

[7] This translates the labels $\mathcal{A}$ computes for nodes in $\mathcal{G}$ to pebbles for the graph $\mathcal{G}^{\times m}$.

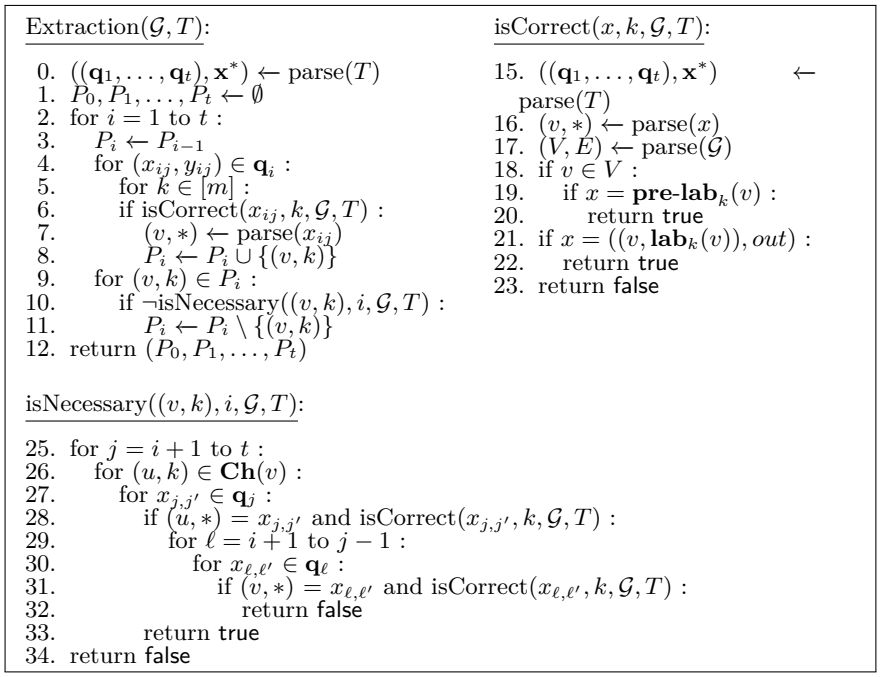

Fig. 5: Algorithm for extracting a pebbling of $\mathcal{G}^{\times m}$ from a transcript of attack in the sMH game.

illegal pebble placement must correspond to an adversary predicting the output to the random oracle. To see how, recall that a node v is added to pebbling step P_i if a correct call has been made in the i-th round of queries. Any correct call for some node v is a query containing $\mathbf{pre\text{-}lab}(v)$, and thus the labels for the parents of v.[8] If the corresponding node was pebbled illegally, one of these labels in $\mathbf{pre\text{-}lab}(v)$ (say for node u) has not been received as a response from the oracle. Such an adversary can be reduced to a predicting algorithm, with success probability bound by Lemma 3. The predictor outputs a guess $\mathbf{lab}(u)$ for the input $\mathbf{pre\text{-}lab}(u)$. For hash-independent adversaries, the hint the predictor receives is the query index $i \in [q]$ of the call which illegally pebbles v. This gives a bound of q/N.

Changes for multiple instances. As described above, when referring to labels and pre-labels we will specify an index $k \in [m]$ to point to the input $\mathbf{x}^*[\mathbf{k}]$ for which a label or pre-label is correct. Additionally, as P is a pebbling for $\mathcal{G}^{\times m}$, when adding pebbles we translate the node indices (for queries labeling distinct instances of a construction), into node indices for (distinct subgraphs of) $\mathcal{G}^{\times m}$.

[8] In the single-instance hash-independent setting, there is a single pre-chosen input x to the construction, so **pre-lab** and **lab** are both defined with respect to x.

Changes for hash-dependence. To extend the argument to a hash-dependent setting, we account for the fact that the adversary $\mathcal{A}$ is free to choose the input value x^*. As $\mathcal{A}$ may not decide upon x^* until its final step, we need to extend the hint to ensure that the predictor is still able to find a correct call for u and *avoid* forwarding the query to its oracle. To do this, we also include in the hint the index of the first correct call for u (if there is one). The additional index required increases the hint space by a factor of q, giving rise to the q^2/N bound from the claim.

Proof. Let bad be the event that P is an illegal pebbling. That is, there is step $P_i \in P$ such that for some $(v,k) \in P_i$ there exists a parent node $\mathbf{Pa}_j((v,k))$, such that $\mathbf{Pa}_j((v,k)) \notin P_{i-1}$. Any node (v,k) from $\mathcal{G}^{\times m}$ is only placed in P if there is a query (in the transcript) of the form $\mathbf{pre\text{-}lab}_k(v)$ where v is the corresponding node in $\mathcal{G}$. If (v,k) is placed illegally, then there is some bad parent node $(u,k) := \mathbf{Pa}_j((v,k))$ such that for some $k \in [m]$, the label $\mathbf{lab}_k(u)$ is contained in $\mathbf{pre\text{-}lab}_k(v)$, though no *correct call* (with respect to the input $\mathbf{x}^*[k]$) was previously made for u in $\mathcal{G}$. Note that if there were a previous correct call for u, the ex-post-facto pebbling algorithm would have kept (u,k) in P_{i-1}.

We next claim that $\Pr[\mathsf{bad}]$ is upper bounded by q^2/N. Let $\mathcal{A}$ be an adversary that triggers bad. We convert $\mathcal{A}$ to an algorithm $\mathcal{B}$ against the prediction game which takes a hint, which is dependent on $\mathcal{A}$, from a space of size q^2. We next apply Lemma 3, to obtain the upper bound. The reduction (which is no longer ex-post-facto) proceeds as follows.

Hint. The hint function is defined as a pair (i,ℓ) where the first entry is the index $i \in [q]$ of the oracle call for the first node (v,k) as defined above (which causes the illegal pebble placement in P). The second is the index $\ell \in [q]$ of the first correct oracle call for the bad parent (u,k) of (v,k) for which there was no *previous* correct oracle call. If there is no correct call for (u,k), the predictor gets the hint (i,i), and if there is no illegal pebble, then the predictor gets the hint $(\perp,\perp)$.

One difference here with that in [3] is the increased hint space. This is because in our setting the predictor does not get access to the input $\mathbf{x}^*$ from the outset – $\mathcal{A}$ outputs $\mathbf{x}^*$ as it finishes. The predictor therefore also needs to receive the index of the query corresponding to the first correct oracle call for the bad parent (u,k) so that it can compute $\mathbf{pre\text{-}lab}_k(u)$. The increase in the hint space changes the bound from q/N to q^2/N.

$\mathcal{B}$ receives a hint (i,ℓ) as defined above and runs $\mathcal{A}$ as follows, aborting if the hint is of the form $(\perp,\perp)$ or is not of the correct form. $\mathcal{B}$ is aiming to predict the value of $\mathsf{H}(\mathbf{pre\text{-}lab}_k(u))$ without querying $\mathbf{pre\text{-}lab}_k(u)$ to the oracle.

Reduction. $\mathcal{B}$ forwards all oracle calls of $\mathcal{A}$ to H up to and including the i-th query. This query is correct, by the definition of the hint function, and so it contains the labels of the parents of v in $\mathcal{G}$, with respect to $\mathbf{x}^*[k]$ for some $k \in [m]$. A correct call for at least one parent, say u, is never queried to H. After receiving the i-th query $\mathcal{B}$ extracts $\mathbf{lab}_k(u)$ from $\mathbf{pre\text{-}lab}_k(v)$. $\mathcal{B}$ then resumes running $\mathcal{A}$ under one of the following cases.

Case 1: If the hint is of the form (i, ℓ) where $i \neq \ell$, then $\mathcal{B}$ forwards all of $\mathcal{A}$'s oracle calls to H until the ℓ-th query and records $\mathbf{pre\text{-}lab}_k(u)$.

Case 2: If the hint is of the form (i, i), then there is no later correct call for u and $\mathcal{B}$ runs $\mathcal{A}$ until it terminates, allowing $\mathcal{B}$ to record $\mathbf{x}^*[k]$ from the adversary's output. $\mathcal{B}$ then uses $\mathbf{x}^*[k]$ to recompute $\mathbf{pre\text{-}lab}_k(u)$, making queries to H as needed.

In both cases, $\mathcal{B}$ has the values for $\mathbf{lab}_k(u)$ and $\mathbf{pre\text{-}lab}_k(u)$, and did not forward $\mathbf{pre\text{-}lab}_k(u)$ to the random oracle. $\mathcal{B}$ now returns $\mathbf{lab}_k(u)$ as its guess for the bits of $\mathsf{H}(\mathbf{pre\text{-}lab}_k(u))$.

If $\mathcal{A}$ triggers the event bad, then $\mathcal{B}$ wins the prediction game. As the hint is from a space of size q^2, and the prediction is in a space of size N, then by Lemma 3 the probability of $\mathcal{B}$ winning the prediction game, and therefore the probability of bad being triggered, is upper bounded by q^2/N. The probability that P is legal is therefore at least $1 - q^2/N$. □

Claim (Pebbling complexity). Let T be a winning transcript corresponding to a q-query pROM adversary $\mathcal{A}$, with fixed random coins and a fixed H, against (q, m)-sMH. Let P be the pebbling extracted from the transcript. Then for any $\lambda \geq 0$:

$$\Pr\left[\exists i \in [t] : |P_i| > \frac{|\sigma_i| + \lambda}{\log N - \log(qm)}\right] < t2^{-\lambda}$$

over the choice of H and the coins of $\mathcal{A}$, where σ_i is the state as defined in Section 2.1.

Changes for multiple instances. An analogous result was proven in [3], for a hash-independent adversary computing a single instance of a construction. Here we adapt the argument to allow for adversaries computing multiple instances of the construction. In particular, the predictor needs to know *for which instance* any given query is correct. Without knowing this, much like in the previous claim, the predictor cannot avoid querying a point for which it would later make a prediction. We provide this information in the hint, which results in a $\log(qm)$ term replacing a $\log(q)$ term in our final bound. This claim and its proof are the same regardless of whether the adversary is hash-independent or not, since it bounds the amount of memory required to store each pebbling set P_i, which intuitively is unaffected by the adversary not being hash-independent.

As each node $(v, k) \in P_i$ is necessary, there is a later round in which there is a correct oracle call (with respect to some $\mathbf{x}^*[k]$) containing $\mathbf{lab}_k(v)$ as a label for one of the input nodes – call this a critical call for v. As there are no correct calls for v with respect to $\mathbf{x}^*[k]$ in the meantime, the adversary making this oracle call must be able to find $\mathbf{lab}_k(v)$ using only the state σ_i. Intuitively the size of σ_i must limit the number of nodes in P_i. This intuition is formalized by constructing an algorithm $\mathcal{B}$ which predicts the label of each node (v, k) in the set P_i with respect to the $(\mathbf{x}^*[k], \mathsf{H})$-labeling of $\mathcal{G}$. The predictor will get a hint from a hint function, and has access to H.

Proof. Let bad_i be the event that for a fixed round $i \in [t]$ and some $\lambda \geq 0$, the size of P_i exceeds the bound in the claim. We claim that $\Pr[\mathsf{bad}_i]$ is upper bounded by $2^{-\lambda}$. Let $\mathcal{A}$ be an adversary which triggers the event bad_i. Given $\mathcal{A}$ we build an algorithm $\mathcal{B}$ against the prediction game, which takes a hint from a space of size at most $q^{|P_i|}2^{|\sigma_i|}$ and successfully predicts, for *each* $(v,k) \in P_i$ the response of oracle query $\mathbf{pre\text{-}lab}_k(v)$, predicting a total of $|P_i| \log N$ bits. Note that k does not have to be the same for each v. Recall a *k-critical call* for node $(v,k) \in P_i$ be a correct oracle call (with respect to $\mathbf{x}^*[k]$) containing $\mathbf{lab}_k(v)$ as a label for one of its input nodes. Intuitively, this is the oracle call that makes (v,k) necessary at step i. The reduction proceeds as follows.

Hint. The hint function is defined as a pair (J, σ_i) where σ_i is the adversary's i-th state and the set $J = \{(j_1,k_1),\ldots,(j_r,k_r)\} \in [q]^r \times [m]^r$ is the set of indices of the critical calls for each $(v,k) \in P_i$, and the indices of the instances for which they are correct. A pair (j,k) means the j-th query from $\mathcal{A}$ is a critical call, correct with respect to $\mathbf{x}^*[k]$.

Algorithm $\mathcal{B}$ receives the hint (as defined above) and runs $\mathcal{A}$ with input σ_i. $\mathcal{B}$ is aiming to predict, for each $(v,k) \in P_i$, the values of $\mathsf{H}(\mathbf{pre\text{-}lab}_k(v))$.

Reduction. $\mathcal{B}$ records all queries from $\mathcal{A}$, and forwards any calls for nodes not in P_i to the oracle, relaying the response. For all critical calls, $\mathcal{B}$ records the labels of the input nodes, along with the instance for which they are correct (these contain the prediction values used later). For all queries of the form $(v, l_1, \ldots, l_\delta)$,[9] $\mathcal{B}$ makes the following *recursive* check for correctness, starting by checking if, for any $k \in [m]$ such that $(v,k) \in P_i$ the query $(v, l_1, \ldots, l_\delta)$ is correct for v. For any query $x_{i,j}$, $\mathcal{B}$ can check if it is correct for some node $w \in \mathcal{G}$, with respect to some $\mathbf{x}^*[k]$, with the following steps

- Check if $x_{i,j}$ is a critical call for some node in P_i (in which case the index of the query appears in J). Then $x_{i,j}$ is correct, as all critical calls are correct by definition. Moreover, the hint contains the instance for which the call is correct.
- Check, recursively, if there are previous correct calls for all parents of w, whose labels match the pre-label in the query. This search ends in at least one critical call. Referring to the hint, $\mathcal{B}$ finds the index for which the call is correct.

If $x_{i,j} = (v, l_1, \ldots, l_\delta)$ is a correct call for $(v,k) \in P_i$, and $\mathbf{lab}_k(v)$ is already recorded, $\mathcal{B}$ responds with $\mathbf{lab}_k(v)$, or else it forwards the call to the oracle and relays the response. If $x_{i,j}$ was a correct call for v with respect to $\mathbf{x}^*[k]$, then $\mathcal{B}$ will record the pair $((v, l_1, \ldots, l_\delta), \mathbf{lab}_k(v))$ to output later. Once $\mathcal{A}$ is run, $\mathcal{B}$ has $\mathbf{lab}_k(v)$ for all $(v,k) \in P_i$ without querying $\mathbf{pre\text{-}lab}_k(v)$ to the oracle. $\mathcal{B}$ can then submit $\mathbf{lab}_k(v)$ as its prediction for $\mathsf{H}(\mathbf{pre\text{-}lab}_k(v))$ for each $(v,k) \in P_i$, correctly predicting the response of the random oracle.

[9] We are considering queries which would label v for any of the m inputs.

Analysis. As the size of the hint space is $q^r m^r 2^{|\sigma_i|}$, and the space of bits being guessed is $N2^{|P_i|}$, then by Lemma 3 $\mathcal{B}$ can do this with probability at most $q^r m^r 2^{|\sigma_i|}/N2^{|P_i|}$.

$$\Pr[\mathcal{B} \text{ wins}] \leq \frac{q^r m^r \ 2^{|\sigma_i|}}{N \ 2^{|P_i|}} \leq \frac{2^{|P_i|\log(qm)} \ 2^{|\sigma_i|}}{2^{|P_i|\log(N)}} = 2^{-|P_i|(\log(N)-\log(qm))+|\sigma_i|} \leq 2^{-\lambda}.$$

Where the second inequality is from $r \leq |P_i|$, and the final one from $-|P_i| < \frac{-|\sigma_i|-\lambda}{\log(N)-\log(qm)}$. This gives $\Pr[\mathsf{bad}_i] \leq \Pr[\mathcal{B} \text{ wins}] \leq 2^{-\lambda}$. Applying the union bound over all time steps gives

$$\Pr[\mathsf{bad}] \leq \sum_i \Pr[\mathsf{bad}_i] \leq t2^{-\lambda}\,.$$

Next, we bound the probability of a collision in the labels $\mathcal{A}$ computes, and then move onto bound the probability $\mathcal{A}$ wins given there are no collisions.

Claim (Collision-free transcript). Let $\mathcal{A}$ be a q-query adversary in the (q,m)-sMH game. Let bad be the event that $\mathcal{A}$ finds a collision in any two labelings $\mathbf{lab}_j$ and $\mathbf{lab}_k$ for some $j,k \in [m]$. Note that the $\mathcal{A}$'s advantage never decreases when there is such a collision, and

$$\mathbf{Adv}^{(q,m)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \Pr[\mathsf{bad}] + \Pr[\mathcal{A} \text{ wins } (q,m)\text{-sMH}_{\mathsf{C}^{\mathsf{H}}} | \neg\mathsf{bad}]\ .$$

To upper-bound the probability of a bad, let $\mathbf{lab}_j(v)$ and $\mathbf{lab}_k(v)$ be the labels for a node v, then a collision in node labels corresponds to a collision in two hash queries to the random oracle H. Namely, if $\mathbf{lab}_j(v) = \mathbf{lab}_k(v)$, then $\mathsf{H}(v,\ldots,\mathbf{lab}_j(\mathbf{Pa}_\delta(v))) = \mathsf{H}(v,\ldots,\mathbf{lab}_k(\mathbf{Pa}_\delta(v)))$. For each node v, let q_v be the number of queries $\mathcal{A}$ makes, beginning with index v. Then, as $\mathcal{A}$ labeled these two nodes, the probability that these two labels collide is at most q_v^2/N. Thus, the probability that $\mathcal{A}$ finds a collision in the labels of any node is at most

$$\frac{1}{N}\sum_1^n q_i^2 \leq \frac{1}{N}\left(\sum_1^n q_i\right)^2 \leq \frac{q^2}{N}\,.$$

The above three claims allow us to complete the proof for Theorem 1. The final steps in the analysis closely follow the proof in [3] for the hash-independent setting.

For any q-query adversary $\mathcal{A}$ in the (q,m)-sMH game, with respect to C^{H}, that produces a collision free transcript T, let P be the ex-post-facto pebbling of $\mathcal{G}^{\times m}$ extracted from T. Let $\mathcal{W}$ be the event that $\mathcal{A}$ wins the game, and assume $\Pr[\mathcal{W}] \geq \epsilon$. Let $\bar{\epsilon} := \log(\epsilon - q^2/N)$, t be the number of steps in P, and $\mathcal{C}$ be the event that P is a legal pebbling *and* that

$$\sum_{i=0}^{t-1} |P_i| \leq \frac{t\bar{\epsilon} + \sum_{i=0}^{t-1}|\sigma_i|}{\log(N)-\log(qm)}\ .$$

By claim 1 (legality), claim 2 (complexity), and an application of the union bound, we have shown that $\Pr[\mathcal{C}] > 1 - (q^2/N) - 2^{-\bar{\epsilon}} = 1 - \epsilon$. Therefore, the probability that $\mathcal{W}$ and $\mathcal{C}$ occur simultaneously is positive, and so an adversary whose transcript satisfies $\mathcal{C}$ can win with probability greater than ϵ. As such a pebbling is legal and complete, the cumulative pebbling cost of P must by definition be lower bounded by $\mathrm{CC}(\mathcal{G}^{\times m})$. Therefore

$$\mathrm{CC}(\mathcal{G}^{\times m}) \leq \frac{-t\log(\epsilon - q^2/N) + \mathrm{CMC}(\mathcal{A})}{\log(N) - \log(qm)} \ .$$

Applying Lemma 2, and rearranging, we obtain the bound for ϵ, and by collecting terms we arrive at the statement in the lemma. □

C Setting P

P is chosen to minimize $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + 2^{-\gamma} + \frac{(2q_{\mathcal{P}}+nm\ell+4q)^2}{N} + \frac{q_{\mathcal{P}}m\ell}{K}$. Since only the first two terms depend on P, it is sufficient to choose P to minimize the sum $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + 2^{-\gamma}$. Note that $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B})$ is non-decreasing in P as having more queries can only increase $\mathcal{B}$'s advantage, and $2^{-\gamma}$ is decreasing in P as the larger P is, the larger γ is by definition. The minimum for the sum occurs when $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B})$ becomes the dominating term. Intuitively, this is the point where the adversary can't compute C^{H} with high probability. As we cannot give a closed form for the exact P that minimizes the sum, we instead provide bounds on P, leveraging the fact that $(1/N)^{m-c} \leq \mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B}) \leq 1$.

We first lower bound P by finding the smallest P s.t. $2^{-\gamma} < 1$. This occurs when $\gamma > 0$, i.e. $((P+1)\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m)) - (\mathrm{CMC}(\mathcal{A}) + \Delta)/t' > 0$, specifically

$$P > \frac{(\mathrm{CMC}(\mathcal{A}) + \Delta)}{\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m))} - 1 \ .$$

We then upper bound P by finding the largest P such that $2^{-\gamma} > (\frac{1}{N})^{m-c}$. This occurs when $-\gamma > -\log N$ i.e. $(P+1)\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m)) - (\mathrm{CMC}(\mathcal{A}) + \Delta) < (m-c)t' \log N$, which in turn rearranges to give

$$P < \frac{(\mathrm{CMC}(\mathcal{A}) + \Delta) + (m-c)t' \log N}{\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m))} - 1 \ .$$

Combining the above gives us bounds on the optimal value P of

$$\frac{(\mathrm{CMC}(\mathcal{A}) + \Delta)}{\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m))} < P + 1 < \frac{(\mathrm{CMC}(\mathcal{A}) + \Delta) + (m-c)t' \log N}{\mathrm{CC}(\mathcal{G})(\log N - \log((q+q_{\mathcal{P}})m))} \ .$$