Multi-Instance Unrecoverability of iMHF-Based Password Hashing *

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

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

Abstract. The study of memory-hard functions (MHFs) has so far focused mainly on provable guarantees on the expected minimum cumulative memory complexity (CMC) required per *evaluation* when amortized over multiple instances. Such results, however, say nothing about whether the passwords in a compromised password bank remain *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 widely accepted security model when inputs have low entropy or are correlated, and requires the adversarial effort to scale linearly with the number of instances broken.

To prove these results, we 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 combine the two resulting frameworks to bound the number of guesses of adversaries with a given *maximum* CMC (over the random oracle and adversary coins) in terms of the pebbling complexity of the graph underlying the iMHF. Combined with known lower bounds on the pebbling complexity of Catena's underlying graph, we obtain concrete unrecoverability bounds for Catena, showing in particular that adversarial advantage diminishes exponentially with the number of instances recovered, with the per-instance advantage growing linearly in the maximum CMC of the adversary.

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

1 Introduction

Password-based cryptography leverages the prevalence of passwords to underpin many applications such as authentication and key derivation. Password storage

* This is the full version of a paper with the same title due to appear at ESORICS 2026.

in such applications is usually protected through salting and hashing, i.e., rather than storing 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 PBKDF2 [14]. However, the ever-growing efficiency with which specialized hardware such as ASICs evaluates hash functions necessitates higher iteration counts that disproportionately affect honest users.

To address this issue, recommendations for password hashing [1] advocate *memory-hard functions* (MHFs) designed so that successfully computing the function requires substantial time and memory. This leverages the fact that dedicated hardware offers far smaller cost savings for memory than for computation. Candidates for MHFs include Scrypt [13], proposed by Percival in 2009 as the first such function, and Argon2 [7], 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 are important when the function is run on a secret input (such as passwords) in an insecure environment, as dMHF memory-access patterns can allow the input to be recovered via side-channel attacks [9]. Constructions of iMHFs are usually based on an underlying *directed acyclic graph* (DAG) that specifies how the hash calls are composed. Such iMHFs are called *graph-based* iMHFs. Examples include Argon2i [7], Catena [11], and Balloon hashing [8].

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 the results with the database entries. Hence, there is an inherent limit to the protection that any construction can provide for *individual* passwords. Despite this, one would still like assurances that the effort needed to recover *multiple* passwords increases, ideally *linearly*, with the number of passwords recovered. The *multi-instance* (mi) security model, introduced by Bellare, Ristenpart, and Tessaro (BRT12) [5], formalizes this requirement in terms of how the adversary's resources grow with the number of "cracked" target hashes. This model was further studied by Farshim and Tessaro (FT21) [10], who proved the efficacy of salting against preprocessing attacks (e.g., rainbow tables [12]) in the multi-instance setting.

In light of ASIC-assisted attacks, measuring attacks by query complexity alone does not suffice. Percival [13] proposed maximum memory usage multiplied by time as an alternative metric. Subsequently, this metric was shown to provide insufficient guarantees against *amortization* [4].[4] The state-of-the-art metric for

[4] Maximum memory usage can be high even when the peak occurs only in the initial stages of an attack, allowing freed memory to be reused for a second instance thereafter.

measuring memory cost, proposed by Alwen and Serbinenko (AS15) [4], is the *cumulative memory complexity* (CMC): the *sum* of memory usage over all time steps of the computation. Alwen and Serbinenko prove lower bounds on the CMC of graph-based iMHFs in terms of the *cumulative complexity* (CC) of the underlying graph, defined as the minimum *pebbling* complexity of the graph [4]. This allows them to prove the non-amortizability of MHF evaluations.

Although provable guarantees on the memory-hardness of graph-based iMHFs give high confidence that *computation* incurs a certain minimum memory cost, they do not translate into provable *unrecoverability* guarantees for secure password storage using such iMHFs. To see this, fix a well-designed MHF F^{H} and 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 provides no 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 achieve an unrecoverability bound of at most the *minimum* of those achieved by $\mathsf{F}_1^{\mathsf{H}}$ and $\mathsf{F}_2^{\mathsf{H}}$.

Multi-instance unrecoverability is the natural security property expected from secure password storage protected by MHFs. Yet the literature contains no formal unrecoverability analysis of MHFs.

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 in terms of the unguessability of the underlying password distribution, the maximum CMC of the adversary (over the random oracle and adversary's coins), and the CC of the graph underlying an iMHF. To this end, we show how to combine the FT21 and AS15 frameworks to obtain unrecoverability bounds for iMHFs. Our proofs bound the number of guesses of any adversary with a given *maximum* CMC in terms of the CC of the underlying graph via an extension of the ex-post-facto pebbling argument of AS15. We then apply the unrecoverability-to-unguessability reduction of FT21 to derive our final bounds.

Along the way, we define *strong* memory-hardness (sMH), whereby, first, an adversary may choose its iMHF inputs adaptively based on the iMHF's underlying hash function, and, second, it may submit an unordered set of candidate input-output pairs, of which a threshold number need be correct. This strengthening is required, along with an extension of the AS15 ex-post-facto pebbling argument and a new unrecoverability-to-unguessability reduction for non-monolithic constructions. We also show a separation between strong memory-hardness and memory-hardness by proving that while ROMix, the core internal construction of Scrypt, is maximally memory-hard, its sMH cost is amortizable, and hence its strong memory-hardness is not optimal. See Appendix A. Despite this, we prove that graph-based iMHFs with a single source and sink that follow AS15 labeling achieve strong memory-hardness.

Future work. We have focused on the security of iMHFs; extending our results to specific or general dMHFs remains open.

2 Preliminaries

Notation. We denote the set of integers from 1 to 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 $v \twoheadleftarrow \mathcal{D}$. 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 (N,M)-random oracle model (RO model or ROM) [6] 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) [4]. 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 x and makes t rounds of batch oracle calls. In each round, $\mathcal{A}$ makes a batch of queries $\mathbf{q}$ and writes to a within-round state st (which does not count toward the adversary's memory usage cost). The batch of responses $\mathbf{y}$ along with st is then given to $\mathcal{A}$, and $\mathcal{A}$ records in a between-rounds state σ_i any information to be 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 distinguished query of the form (l, out), where out is a special symbol. This query is neither counted toward $\mathcal{A}$'s query complexity nor answered by the oracle, but is recorded in the transcript. When $\mathcal{A}_*^{\mathsf{H}}$ terminates, $\sigma_{\mathcal{O}}$ is returned. The structure of a pROM algorithm is shown in Fig. 1 (top left).

Password samplers and salt generators. A password sampler $\mathcal{P}_m$ in BRT12/FT21 is a randomized algorithm that takes no input and outputs a vector of m passwords $\mathbf{pw}$ together with possible leakage z on the passwords. The leakage z models the information the adversary holds about the password distribution prior to its attack. Salts are generated by a salt-generator algorithm Gen, a randomized algorithm that takes as input a user index $i \in [m]$ and a counter $j \in [\ell]$, and outputs salts $\mathbf{sa}[i,j]$ (allowing a password to be reused across ℓ salts) from a space of size K, which we consider to be $[K]$. We denote a generator sampling uniformly at random from a space of size K as $\mathsf{Gen} = [K]$. When no salting is used, Gen outputs m empty salts. We write $\mathsf{Gen} = \epsilon$ and assume $\ell = 1$.

Unguessability and salted unguessability. We briefly recall the notions of (salted) unguessability from [5,10]. 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 such game is the GUESS game from [5]. An adversary

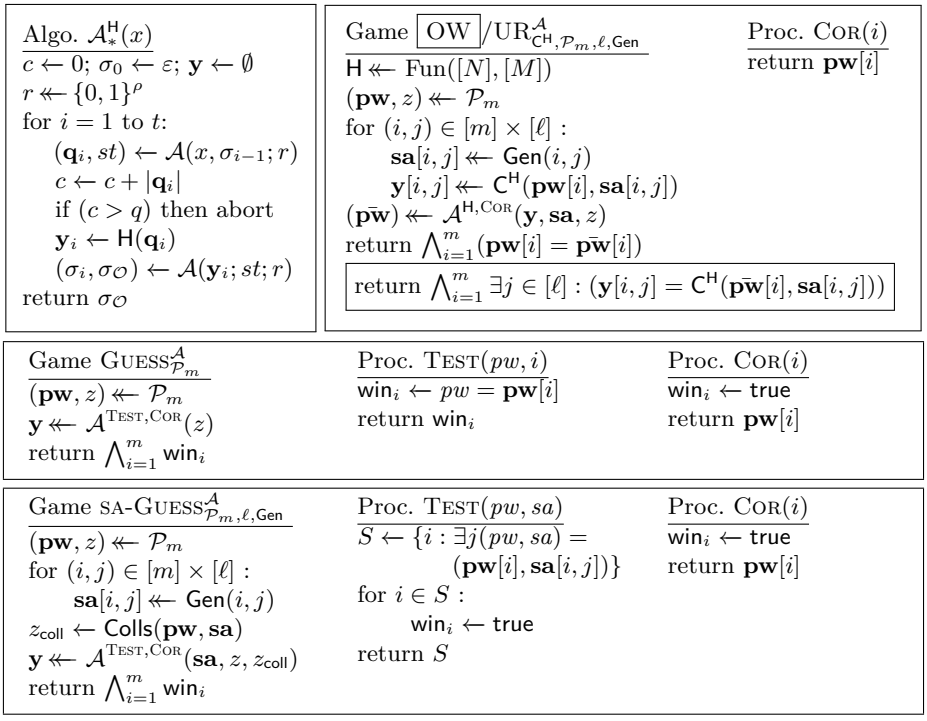


Fig. 1: Top left: Structure of an algorithm in pROM. Top right: UR and OW games w.r.t. a m-sampler $\mathcal{P}_m$ and salt generator Gen. Middle: Guess game w.r.t. $\mathcal{P}_m$. Bottom: sa-Guess game w.r.t. $\mathcal{P}_m$. Here z_{coll} denotes the collision pattern of password-salt pairs.

$\mathcal{A}$ in the Guess game is given leakage z and has 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. We consider adversaries that make at most c queries to Cor, i.e., they automatically "win" on c of the passwords. We refer to an adversary $\mathcal{A}$ with T queries to Test and c queries to Cor as a (T,c)-query adversary, and when $c = 0$ simply as a T-query adversary. For an adversary $\mathcal{A}$ and a password sampler $\mathcal{P}_m$ in $\textsc{Guess}^{\mathcal{A}}_{\mathcal{P}_m}$, we formally define $\mathbf{Adv}^{\textsc{Guess}}_{\mathcal{P}_m}(\mathcal{A}) := \Pr[\textsc{Guess}^{\mathcal{A}}_{\mathcal{P}_m}]$.

For the salted setting, we again consider adversaries that 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 every password-salt pair in the sampled set, and win flags are returned for every password whose salt also matches. 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 the 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}. The pattern is an $m\ell \times m\ell$ matrix with entries

$((i_1, j_1), (i_2, j_2))$ set to 1 if and only if $(\mathbf{pw}[i_1], \mathbf{sa}[i_1, j_1]) = (\mathbf{pw}[i_2], \mathbf{sa}[i_2, j_2])$. A (T, c)-query adversary and its advantage are defined analogously to the unsalted case: $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{A}) := \Pr[\text{SA-GUESS}^{\mathcal{A}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}]$.

Unrecoverability and one-wayness. We recall the notions of multi-instance unrecoverability and one-wayness in relation to password hashing; the security games for both are given in Fig. 1. Unrecoverability 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 $\mathbf{y}$ together with $\mathbf{sa}$ and the leakage z. For this game, the collision pattern of password-salt pairs is public and does not need to be provided to $\mathcal{A}$. The adversary $\mathcal{A}$ wins if it recovers all m original passwords. Formally, for a hash-based construction C^{H}, password sampler $\mathcal{P}_m$ and salt generator Gen, we define the advantage of adversary $\mathcal{A}$ in $\text{UR}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}$, presented in Fig. 1 (top right), as $\mathbf{Adv}^{\text{UR}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{A}) := \Pr[\text{UR}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}]$. The one-wayness game is similarly defined, except that the adversary wins if it outputs values that, when evaluated under C^{H}, result in the same challenge vector $\mathbf{y}$, and for the $\text{OW}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}$ game presented in Fig. 1 (top right, boxed), we define $\mathbf{Adv}^{\text{OW}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{A}) := \Pr[\text{OW}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}]$. When Gen outputs the empty string ϵ, and $\ell = 1$, we omit these subscripts from the advantage.

2.2 Graph-based iMHFs and cumulative memory complexity

Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG), where the elements of the node set $V = [n]$ are 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 $\text{indeg}(i)$ be the in-degree of node i and $\delta := \max_i\{\text{indeg}(i)\}$. We consider iMHFs that use the labeling scheme from AS15 [4]. We reproduce the definition of the labeling here and extend it to allow for salting. To this end, the initial label l is drawn from $L \times [K]$ rather than L (i.e., l takes the form (pw, sa)).

(H, l)-labeling of graphs [4]. 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 $\mathbb{H} := \text{Fun}(V \times L^{\delta} \times [K], L)$. 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:

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

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 computes

the label. For clarity, however, we omit this padding function. We use $\mathbf{pre\text{-}lab}(v)$ to refer to the H inputs that define v's label, i.e., $\mathbf{pre\text{-}lab}(v) = (v, l)$ if $\mathrm{indeg}(v) = 0$, and $\mathbf{pre\text{-}lab}(v) := (v, \mathbf{lab}(\mathbf{Pa}_1(v)), \ldots, \mathbf{lab}(\mathbf{Pa}_d(v)))$, where $d = \mathrm{indeg}(v)$, otherwise. Note that as l is a password-salt pair, a salt is only included when $\mathrm{indeg}(v) = 0$. Using this labeling, a graph-based iMHF C^{H} is defined as follows.

Graph-based iMHF [4]. Let $\mathcal{G} = (V, E)$ be a DAG on n vertices, with maximum in-degree δ, a single source node 1, and a single sink node n. 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 oracle function family $\mathsf{C} = \mathsf{C}^{\mathbb{H}}_{\mathcal{G}}$, 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 $\mathsf{C}^{\mathsf{H}}(l) := \mathbf{lab}(n)$, where $\mathbf{lab}$ is the (H, l)-labeling of $\mathcal{G}$.

We primarily consider the labeling set $L := [N]$, but the results hold for any labeling set L with $|L| = N$ that can be efficiently mapped to $[N]$. We follow the notation of [4] for multiple instances of a graph. That is, for a DAG $\mathcal{G} = (V, E)$ and a positive integer m, we use $\mathcal{G}^{\times m}$ 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.

To model the memory required 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 the simple rules below. 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 of the graph can be pebbled at any time. A pebble may be placed on any other node only if all of its parents had pebbles on them in the previous round. We consider the *parallel* pebbling game, in which 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 be added only if its predecessors have pebbles on them 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 sets of source and sink nodes, respectively. Note 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 has at most q pebbles to place across at most t rounds, we define the cumulative pebbling complexity (CPC) of $\mathcal{A}$'s pebbling of $\mathcal{G}$ as $\mathrm{CPC}(\mathcal{A}) := \sum_{i=0}^{t} |P_i|$. We define the q-CPC of $\mathcal{G}$ as the minimum CPC of any q-pebble adversary's legal and complete pebbling of $\mathcal{G}$, i.e., if $S[q]$ denotes the set of all q-pebble adversaries, then $\mathrm{CPC}_q(\mathcal{G}) := \min_{\mathcal{A} \in S[q]}(\mathrm{CPC}(\mathcal{A}))$. Beyond a sufficiently large q

(e.g., $q > |V|^2$), $\mathrm{CPC}_q(\mathcal{G})$ 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, denoted $\mathrm{CC}(\mathcal{G})$.

Cumulative memory complexity. We consider memory-constrained adversaries in the 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 us to analyze the total memory used by a parallel adversary over its execution. We define the *maximum* cumulative memory complexity of an adversary $\mathcal{A}$ with respect to x as

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

where the maximum is taken over the choice of random oracle H and random coins r. For our applications, the input to $\mathcal{A}$ is the empty string, so we omit x and write $\mathrm{CMC}_{\max}(\mathcal{A})$. Note that Alwen and Serbinenko [4] define CMC as the *expected* value over the random oracle H and randomness r. Maximum CMC upper bounds expected CMC, and thus advantage bounds based on maximum CMC are usually weaker than those stated in terms of expected CMC. It is, however, well motivated in practice: the cost of a brute-force attack on dedicated hardware is governed by the *peak* memory the circuit must physically provide, rather than the average across runs. Moreover, it is compatible with the standard worst-case complexity metrics in cryptography, such as space and time.[5]

Single-sink vs. multi-sink graphs. Current metrics for CMC assume that a full computation of a construction must be performed and that partial computations are of no use to an adversary. For the purposes of guessing passwords, this assumption is false for both expected and maximum CMC. Consider a function F^{H} with a single sink that has a high CMC, and define a construction $\mathsf{C}^{\mathsf{H}} := \mathsf{F}^{\mathsf{H}}(x)||\mathsf{H}(x)$ with two sinks. Now, C^{H} still has a high CMC. However, to check whether a guessed password x' is correct, it is sufficient to consider the second sink only, since $\mathsf{H}(x) \neq \mathsf{H}(x')$ already shows that $x \neq x'$. Hence, the adversary needs to make only a single hash query for such a check rather than comparing the full computation of $\mathsf{C}^{\mathsf{H}}(x')$ with $\mathsf{C}^{\mathsf{H}}(x)$. Note that this holds not only for unrecoverability but also for finding collisions, since non-collision is easy to verify without a full computation. While our example is somewhat 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 all major graph-based iMHFs, including those following the AS15 labeling, e.g., Catena [11], and those that do not, e.g., Argon2i [7] and Balloon hashing [8], adhere to this restriction and moreover have a single source node.

[5] An algorithm is poly-time in the RO model if it runs in polynomial time in the length of its input for *any* choice of the oracle and random coins.

Game m-MH$^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}$	Game (τ, Q)-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 $(\lvert\{\mathbf{x}\}\rvert > Q) \vee (\lvert\{\mathbf{y}\}\rvert > Q)$:
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 $\lvert S\rvert \geq \tau$

Fig. 2: The hash-independent (left) and 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).

3 Strong Memory Hardness

In the existing hash-independent definition of memory-hard functions, a parallel adversary chooses the input x to the function at the outset, and the function is deemed memory hard if the adversary needs a large CMC to evaluate the function correctly (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). The game is played by two-stage adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$. 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, we define $\mathbf{Adv}^{m\text{-MH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) := \Pr[m\text{-MH}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}]$ and say the construction C^{H} is (m, q, ϵ, μ)-memory hard if for all q-query adversaries $\mathcal{A}$ with $\mathrm{CMC}_{\max}(\mathcal{A})$ at most μ, $\mathbf{Adv}^{m\text{-MH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \epsilon$.

We introduce a stronger notion, the *strong* MH game sMH, that extends adversarial capabilities in two orthogonal ways. First, it grants the adversary oracle access while choosing inputs. Second, it allows the adversary to output an *unordered* set of Q inputs and Q outputs, of which only $\tau \leq Q$ need be correct MHF computations. This game is formalized in Fig. 2 (right). We define $\mathcal{A}$'s advantage in the sMH game as $\mathbf{Adv}^{(\tau,Q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) := \Pr[(\tau, Q)\text{-sMH}^{\mathcal{A}}_{\mathsf{C}^{\mathsf{H}}}]$. Strong $(\tau, Q, q, \epsilon, \mu)$-memory hardness is defined analogously.

This definition is strictly stronger than memory-hardness: an MHF with high (even optimal) $\mathrm{CMC}_{\max}$ can have fast evaluations, i.e., non-optimal strong memory hardness. To prove this separation, we show in Appendix A that while Scrypt is provably maximally memory-hard [3], this is not the case in the strong memory-hardness game.

We extend the AS15 results and show that single-sink, single-source *graph-based* MHFs that follow AS15 labeling are strongly memory hard. To this end,

we adapt the ex-post-facto pebbling argument. We first give an 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 CMC close to that of any adversary that simply pebbles each instance of the graph. This result is proved by applying three auxiliary lemmas to the definition of the amortized cost. The first ([4, Lemma 11]) calculates the expected savings an adversary can make by exploiting collisions between the labelings 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 $\mathcal{G}^{\times m}$, whose CC is lower bounded in [4, Lemma 12]. The CMC of an adversary computing a single instance of $\mathcal{G}^{\times m}$ is bounded, roughly, by $\mathrm{CC}(\mathcal{G}^{\times m})$. 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 ([4, Lemma 3]) proves that the CC of two node-disjoint graphs is additive. The bound for the one-off cost of $\mathcal{G}^{\times m}$ can then be given in terms of that of $\mathcal{G}$. Together, in [4, Theorem 3], these lemmas bound the amortized cost in terms of $\mathrm{CC}(\mathcal{G})$.

Extension to strong memory-hardness. In the proof below, we adapt the techniques of [4, Lemma 12] to the (τ, Q)-sMH game, and derive an upper bound on the advantage of any adversary in this game based on *maximum* CMC.

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

$$\mathbf{Adv}^{(\tau,Q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \;\leq\; \frac{2q^2}{N} + t\,2^{-\gamma_{\max}} \;,$$

where $\gamma_{\max} := \frac{\tau \cdot \mathrm{CC}(\mathcal{G})\,[\log N - \log(q\tau)] - \mathrm{CMC}_{\max}(\mathcal{A})}{2t}$.

Intuitively, $\gamma_{\max}$ captures the gap between the maximum CMC of an adversary computing τ instances following the canonical pebbling algorithm, where for each pebble, the adversary queries the hash to compute the next label, with $\log(q\tau)$ accounting for the expected saving due to higher query complexity, and the actual $\mathrm{CMC}_{\max}$ of $\mathcal{A}$, averaged over t steps (and halved for technical proof reasons). Thus, $\gamma_{\max}$ becomes large as the adversarial $\mathrm{CMC}_{\max}$ falls below the minimum required for the construction. We give the full proof in Appendix B.

In our extension, we must 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 extended hint contains the position in the transcript, if present, of the query for which the adversary will predict the output.

We make a further extension to analyze the pebbling complexity, in which we again 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, which now 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 an adversary's advantage 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 establishes security bounds in terms of the adversary's $\mathrm{CMC}_{\max}$ (as opposed to query complexity only) and relies on the upper bounds established in the previous section for computing an iMHF on τ distinct hash-dependent inputs.

Theorem 2 (UR security of C^{H}). *Let $K, N, P, q, n, m, c, t, \delta, \ell \in \mathbb{N}$, $P < q$, $\mathcal{P}_m$ be an 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 $\mathcal{A}$ there is a P-query c-corruption $\mathcal{B}$ against* SA-Guess *such that*

$$\mathbf{Adv}^{\mathrm{UR}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{A}) \;\le\; \mathbf{Adv}^{\text{SA-Guess}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + t\,2^{-\gamma_{\max}} + \frac{(nm\ell+4q)^2}{N} + \frac{m\ell}{K},$$

where $\gamma_{\max} = \frac{(P+1)\,\mathrm{CC}(\mathcal{G})[\log N - \log(q(P+1))] - (\mathrm{CMC}_{\max}(\mathcal{A}) + \mathrm{CMC}_{\max}(\mathcal{P}_m) + mt\log N)}{2t}$.

In Appendix C, we derive lower and upper bounds for P and show that the value of P that minimizes the right-hand side of the bound in the theorem has the form $\Theta\left(\frac{\mathrm{CMC}_{\max}(\mathcal{A})}{\mathrm{CC}(\mathcal{G})\log N}\right)$.

Technically, password samplers that make no random oracle queries can be rewritten to have a CMC of zero, as such samplers do not need to pass state between rounds of random oracle queries. This is an artifact of the definition used in the ROM; in contrast, in the standard model, samplers must maintain state across computational steps. For this reason, we keep the $\mathrm{CMC}_{\max}(\mathcal{P}_m)$ term above.

We start with the multi-instance unrecoverability game and then move to a game that lazily samples the RO, setting a bad flag and aborting on collisions in the outputs. By aborting on collisions, we stop the adversary from choosing distinct inputs that collide early on in the computation, which would otherwise make the computation easier to amortize. We then transition to a game that sets a bad flag and aborts if the adversary makes a query for an index > 1 that was already 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 make a series of transitions after which the game no longer stores intermediate points generated during the challenge generation phase. Finally, we transition to a game that aborts if the adversary computes C^{H} on more than P distinct inputs, which we justify via a reduction to the $(P+1, q)$-sMH game defined in Section 3. Having bounded the number of full computations the adversary can perform, we reduce to the sa-Guess game, where the adversary may 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 a 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, $\Pr[\mathrm{G}_0] = \mathbf{Adv}^{\mathrm{UR}}_{\mathsf{C}^{\mathsf{H}},\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{A})$.

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 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) makes a total of at most $2q + mn\ell$ 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 pre-label generated during the challenge phase, without first completing the computation from the corresponding input, and then aborts after bad_2 is set. This check is performed by a procedure called findChain. Given a pre-label y for a node v, findChain checks if there exists a computation chain from an initial input up to the node v and returns either that input or $\perp$. On input y, 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 at most one inverse y'. If no inverse exists, the function returns $\perp$; otherwise, it recursively applies findChain on y' for $\mathbf{Pa}_i(v)$. When findChain is

called on a pre-label on node 1, it will be in the form $(1, l)$, where $l = (pw, sa)$; hence 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 $\mathsf{C}^{\mathsf{H}}(pw, sa)$ assign to each node v a label l. If $\mathcal{A}$ makes a query containing the label l without fully computing $\mathsf{C}^{\mathsf{H}}(pw, sa)$ up to the corresponding node v, then bad_2 is set. We show in the challenge point prediction claim below that $\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq \Pr[\mathsf{bad}_2] \leq m\ell q/N$.

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; otherwise, it samples a fresh random value. The bad_1 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_2 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_6: This game sets a bad flag bad_3 whenever the adversary fully computes C^{H} on more than P distinct inputs, and then aborts when bad_3 is set. Games G_6 and G_5 are identical until bad_3. We show, via a reduction to a strong $(P + 1, q)$-sMH game, in the bounded chain completion claim below that $\Pr[\mathrm{G}_5] - \Pr[\mathrm{G}_6] \leq \Pr[\mathsf{bad}_3] \leq \mathbf{Adv}^{(P+1,q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{B})$. We conclude the proof by showing in the G_6 to SA-GUESS reduction claim below that the advantage of any adversary in G_6 is bounded by that of an adversary in the SA-GUESS game.

Claim (Collisions). For the parameters 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 standard bound on the probability that two hash outputs collide and is omitted for brevity.

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 standard collision bound on the probability that any of the $m\ell$ challenge points collide with any of the queries of $\mathcal{A}$ and is omitted for brevity.

Claim (Bounded chain completion). For the parameters above:

$$\Pr[\mathcal{A} \text{ sets } \mathsf{bad}_3 \text{ in } \mathrm{G}_6] \leq \mathbf{Adv}^{(P+1,q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{B}) \leq t\, 2^{-\gamma_{\max}} + \frac{2q^2}{N},$$

where $\gamma_{\max} := \big((P{+}1)\mathrm{CC}(\mathcal{G})[\log N - \log(q(P+1))] - \mathrm{CMC}_{\max}(\mathcal{B})\big)/(2t)$ and $\mathrm{CMC}_{\max}(\mathcal{B}) = \mathrm{CMC}_{\max}(\mathcal{A}) + \mathrm{CMC}_{\max}(\mathcal{P}_m) + mt \log N$.

Proof. Given an adversary $\mathcal{A}$ as defined in the theorem statement, we construct a q-query adversary $\mathcal{B}$ against $(P{+}1, q)$-sMH, where $\mathrm{CMC}_{\max}(\mathcal{B}) = \mathrm{CMC}_{\max}(\mathcal{A}){+}$ $\mathrm{CMC}_{\max}(\mathcal{P}_m) + mt \log N$. First, we reverse the game hops. Algorithm $\mathcal{B}$, with access to H, samples a random password vector and leakage z from $\mathcal{P}_m$, 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, with source-query inputs forming $\mathbf{x}$ and sink-query responses forming $\mathbf{y}$.

To handle $\mathcal{A}$'s corruption queries, $\mathcal{B}$ runs $\mathcal{P}_m$ and stores the password vector $\mathbf{pw}$. This increases $\mathrm{CMC}_{\max}(\mathcal{B})$ by $\mathrm{CMC}_{\max}(\mathcal{P}_m) + mt \log N$. 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 $\mathrm{CMC}_{\max}$ cost. If $\mathcal{A}$ triggers bad_3, then there exists a set of $(P+1)$ correct input-output pairs in $\sigma_{\mathcal{O}}$, and $\mathcal{B}$ wins the $(P+1, q)$-sMH game. □

Claim (G_6 to sa-Guess *reduction).* Let $K, N, P, q, n, m, c, \delta, \ell \in \mathbb{N}$, $\mathcal{G}$ 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 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 such that

$$\Pr[\mathrm{G}_6] \leq \mathbf{Adv}^{\text{sa-Guess}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B})\ .$$

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 $\mathsf{C}^{\mathsf{H}}(x)$. [6] $\mathcal{B}$ receives the salt vector $\mathbf{sa}$ and 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.

[6] For any $\mathcal{A}$ that returns x without computing $\mathsf{C}^{\mathsf{H}}(x)$ we can create an $\mathcal{A}'$ that runs $\mathcal{A}$ and computes $\mathsf{C}^{\mathsf{H}}(x)$ before returning x.

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 they do, $\mathcal{B}$ queries (x, sa) to TEST. If TEST returns a non-empty set, $\mathcal{B}$ responds to $\mathcal{A}$'s query with a value $y \in \mathbf{y}$ corresponding to (x, sa) for one of the indices in S. Otherwise, it lazily samples the random oracle output.

As we have a single sink, any query to the sink node must correspond to a full computation. Note that if there were multiple sink nodes, then a query to TEST would be made any time a sink node is reached, regardless of whether the other sinks have been reached, i.e., regardless of whether the whole computation has been completed. $\mathcal{A}$ could have computed the function for at most P distinct start points; otherwise, bad_3 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, 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 does not fully complete the computation, we do not set a bad flag. As $\mathcal{B}$ will have returned y if TEST returns true, whenever $\mathcal{A}$ wins, $\mathcal{B}$ will also win. □

Combining the game transitions above gives

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

where the last inequality is due to $(nm\ell+4q)^2 \geq 2q^2 + m\ell q + (nm\ell+2q)^2$ and the definition of $\gamma_{\max}$ as per the statement of Theorem 2. □

One-wayness. Password-storage security can also be analyzed with respect to a standard one-wayness notion, in which the adversary's goal is to recover *any* input as long as it produces the same construction output as that of the actual password. One can move between unrecoverability and one-wayness straightforwardly 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., for any OW adversary $\mathcal{A}$, there exists a UR adversary $\mathcal{B}$ such that

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

This bound is sufficient for password hashing, as passwords typically have low entropy and thus the UR bound dominates the term due to hash collisions. This means that even if the format of the passwords is not checked and a password-based system accepts any value as input, security remains intact.

Interpretation. Consider a setting without corruptions, i.e., $c = 0$. Setting P as discussed in Appendix C, will make $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(P)$ the dominating term in the bound in Theorem 2. Assuming a uniform salt generator $\mathsf{Gen} = [K]$ and moving from SA-GUESS to GUESS by applying [10, Thm 3], we have the bound $\mathbf{Adv}^{\text{SA-GUESS}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(P) \leq \mathbf{Adv}^{\text{GUESS}}_{\mathcal{P}_m}(P) + m^2\ell^2/K$, which is dominated by the first term on the right-hand side for a sufficiently large K. Applying [10, Thm 2] gives the bound $\mathbf{Adv}^{\text{GUESS}}_{\mathcal{P}_m}(P) \leq \binom{P}{m}\mathbf{Adv}^{\text{GUESS}}_{\mathcal{P}_m}(m)$. For a password sampler $\mathcal{P}_m = [D]_m$ that outputs m passwords uniformly chosen from a dictionary of size D, the advantage of a P-query adversary in the GUESS game is at most $\binom{P}{m}/D^m \leq \left(\frac{eP}{mD}\right)^m$. Hence, for any graph-based iMHF that follows AS15 labeling, we can upper bound the advantage as

$$\mathbf{Adv}^{\text{UR}}_{\mathsf{CH},[D]_m,\ell,[K]}(P) \leq \tilde{\mathcal{O}}\left(\left(\frac{\text{CMC}_{\max}(\mathcal{A})}{\text{CC}(\mathcal{G})mD}\right)^m\right) .$$

This matches the intuition that for salted passwords, the adversary has to spread its memory usage across all m instances, using $\text{CMC}_{\max}/m$ per instance. So, for each instance, it can compute the construction $\text{CMC}_{\max}/(m\text{CC}(\mathcal{G}))$ times, and each attempt is a correct guess with probability $1/D$. As the adversary needs to guess correctly for m independent instances, we have the bound above. Hence, the adversarial effort to recover the passwords scales linearly in both the CC of the underlying graph and the number of passwords. Note that neither ℓ nor K appears in this simplified bound, as they only contribute to an additive term that is dominated.

Replacing $\text{CC}(\mathcal{G})$ with known bounds on the cumulative complexity of the underlying graphs for iMHFs gives us concrete relationships between adversarial CMC and unrecoverability advantage. Specifically, Catena is a graph-based iMHF that follows AS15 labeling, and it has been shown that the cumulative complexity of its underlying graph is lower-bounded by $\tilde{\Omega}(n^{1.5})$ and upper-bounded by $\tilde{\mathcal{O}}(n^{1.625})$ [2]. Therefore, for Catena we have the following concrete bound, which can be used to set the scheme's parameters so that a given adversarial memory budget yields a target unrecoverability advantage:

$$\mathbf{Adv}^{\text{UR}}_{\mathsf{Catena},[D]_m,\ell,[K]}(P) \leq \tilde{\mathcal{O}}\left(\left(\frac{\text{CMC}_{\max}(\mathcal{A})}{n^{1.5}mD}\right)^m\right) .$$

Acknowledgments. Pooya Farshim was supported in part by EPSRC grant EP/V034065/1. We thank an anonymous TCC'26 reviewer for in-depth comments that highlighted a gap in an earlier version regarding the distinction between expected and maximum CMC. Their further comments improved the paper throughout.

References

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

2. Alwen, J., Blocki, J., Pietrzak, K.: Depth-robust graphs and their cumulative memory complexity. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 3–32. Springer, Cham (Apr / May 2017). https://doi.org/10.1007/978-3-319-56617-7_1
3. 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
4. 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
5. 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
6. 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
7. 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)
8. 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
9. 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
10. 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
11. 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
12. 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
13. Percival, C., Josefsson, S.: The scrypt Password-Based Key Derivation Function. RFC 7914, IETF (2016), https://www.rfc-editor.org/info/rfc7914
14. PKCS #5: Password-based cryptography specification. RSA Data Security, Inc. (Sep 2000), version 2.0

A Amortizing Scrypt over Hash-Dependent Inputs

Here, we give a separation between the hash-independent and hash-dependent notions of CMC. We first define what it means for a construction to have a high CMC in our stronger setting, in which 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 τ instances of the construction C^{H} if $\mathbf{Adv}^{(\tau,\tau)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \geq \epsilon$. Given a construction C^{H}, we define its *hash-dependent* (q,ϵ)-$\mathrm{CMC}_{\max}$ as the $\mathrm{CMC}_{\max}$ of the "best" q-query algorithm that computes C^{H} on some (typically the easiest) input with probability at least ϵ. That is,

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

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 $\tau \in \mathbb{N}$, we define the (τ,q,ϵ)-$\mathrm{CMC}_{\max}$ of C^{H} as the CMC of the best q-query algorithm that computes C^{H} on at most τ (possibly hash-dependent) instances with probability at least ϵ, where the effort is scaled by the number of instances computed. In other words,

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

where $S[q,\epsilon,\bar{\tau},\mathsf{C}^{\mathsf{H}}]$ is the set of all q-query algorithms $\mathcal{A}$ in the pROM that have advantage at least ϵ in the $(\bar{\tau},\bar{\tau})$-sMH game against C^{H}. We say that a construction C is (τ,q,ϵ,μ)-strongly-memory-hard if its (τ,q,ϵ)-$\mathrm{CMC}_{\max}$ is at least μ. Note that when $\tau = 1$, 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, this metric reflects any savings in CMC, even if they are the result of sharing memory across computations for (possibly many) different instances.

Separation. In [3], 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 the same 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) $\mathrm{CMC}_{\max}$ 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 $\mathrm{CMC}_{\max}$ of $\Omega(n^2 \log N)$ per instance. The upper bounds $\mathcal{O}(n^2 \log N)$ and $\mathcal{O}(n \log N)$ hold on every run (the attack is deterministic given H), and the hash-independent lower bound $\Omega(n^3 \log N)$ of [3] is likewise a worst-case/per-run statement. Hence, the separation stands unchanged under the worst-case metric we adopt.

Note that the sCMC definition extends adversarial capabilities in two ways. The first is hash-dependent inputs; the second is the unordered output set. Importantly, the attack in this section leverages only 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 in [3], we focus on the ROMix construction. ROMix is defined in terms of an integer n and a hash function H, which we model as a random oracle in the (N^2, N)-RO model, 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.

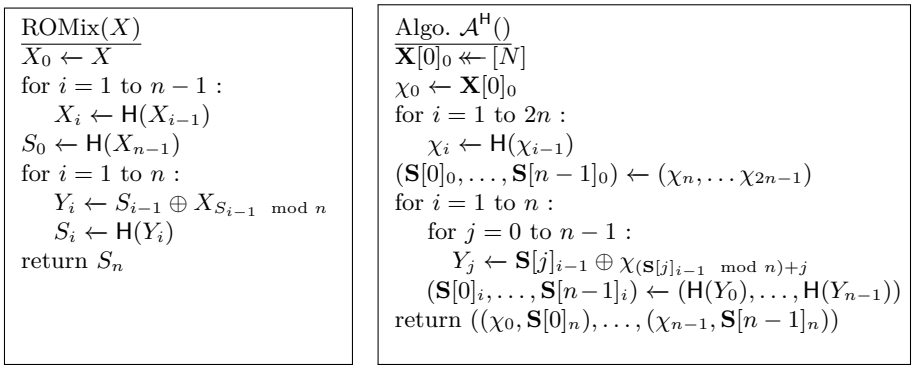


Fig. 3: Left: The ROMix construction. Right: An algorithm that computes ROMix on n inputs, with an sCMC that is n times lower than the sCMC required for a single instance.

ROMix comprises 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 $\mathrm{CMC}_{\max}$ of $\mathcal{O}(n^2 \log N)$, as each stored value requires $\log N$ bits. Alwen et al. [3] show that this is essentially optimal.

More precisely, in [3], the authors show that any algorithm computing Scrypt on a vector of n inputs chosen independently of the hash function will (with overwhelming probability) require an overall $\mathrm{CMC}_{\max}$ of $\Omega(n^3 \log N)$ and hence an amortized $\mathrm{CMC}_{\max}$ 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 $n/(2p)$ iterations from the nearest stored label. The required $\mathrm{CMC}_{\max}$ 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 computing Scrypt on the inputs $\mathbf{X} = (X, \mathsf{H}(X), \mathsf{H}^2(X), \ldots, \mathsf{H}^{n-1}(X))$ needs only an overall sCMC of $\mathcal{O}(n^2 \log N)$ and hence an amortized sCMC of just $\mathcal{O}(n \log N)$ per instance.

Intuitively, the algorithm we propose chooses inputs that lead to overlaps in the hash chains computed by 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 across 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 denote, respectively, the values X_j and S_j computed for the input $\mathbf{X}[i]$. We consider algorithms 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, 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 also give less formal pseudo-code in Fig. 3 and describe it here.

The algorithm computes $\mathbf{X}[1]_0$ to $\mathbf{X}[1]_{2n-1}$ from the first input, as the other $n-1$ inputs are generated by hashing it; both these inputs and their corresponding $\mathbf{X}[i]_0$ to $\mathbf{X}[i]_{n-1}$ are contained within $\mathbf{X}[1]_0, \ldots, \mathbf{X}[1]_{2n-1}$. For computing $\mathbf{S}[i]_0, \ldots, \mathbf{S}[i]_{n-1}$ the algorithm follows the canonical approach.

More formally, for rounds $i = 0$ to $2n-1$, algorithm $\mathcal{A}$ outputs the pair $(\chi_i, \mathbf{q}_i) = (\chi_{i-1} || \mathsf{H}(X_{i-1}), X_i)$. After these $2n$ rounds, χ_{2n-1} contains $X, \mathsf{H}(X), \ldots, \mathsf{H}^{2n-1}(X)$. Note that $|\chi_{2n-1}| = 2n \log N$, and across these $2n$ rounds $\text{sCMC}(\mathcal{A}) = (2n)(2n-1)\log N/2 \leq 4n^2 \log N$, 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 considerable overlap, e.g., $\mathbf{X}[0]_{n-1} = \mathbf{X}[1]_{n-2} = \mathbf{X}[2]_{n-3} = \ldots = \mathbf{X}[n-1]_0$.

For each round $i = 2n$ to $3n-1$, $\mathcal{A}$ computes S_i for each input X in parallel. Specifically, at the end of each round i, $\mathcal{A}$ outputs the pair $(\chi_i, \mathbf{q}_i) = (\chi_{2n}, \mathbf{S}[0]_{i-1} \oplus \chi_{(\mathbf{S}[0]_{i-1} \bmod n)}, \ldots, \mathbf{S}[n-1]_{i-1} \oplus \chi_{(\mathbf{S}[j]_{i-1} \bmod n)+n-1})$.

For each of these rounds, the size of (χ_i) is $2n \log N$, giving $\text{sCMC}(\mathcal{A})$ for rounds $i = 2n+1$ to $3n$ as $2n^2 \log N$.

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 $\text{sCMC}(\mathcal{A})$ is $7n^2 \log N$, which, when amortized over the n values being computed simultaneously, gives an amortized $\text{sCMC}(\mathcal{A})$ 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, for such k, we obtain an amortized sCMC of $o(n^2 \log N)$, which is strictly 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 $o(n^2 \log N)$.*

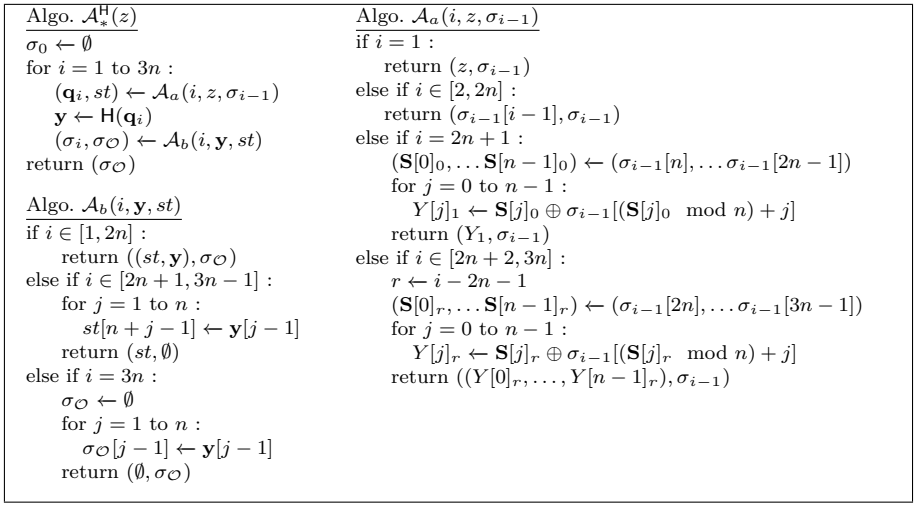


Fig. 4: A parallel algorithm written in the style of a pROM algorithm that computes ROMix on n inputs, with an amortized sCMC that is n times lower than the sCMC required for a single instance.

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

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 $\text{sCMC}(\mathcal{A}) = o((n^2 + n + k) \log N) = o(n^2 \log N)$.

This shows that even for small values of k, as long as $k = \omega(1)$, there is an amortizing algorithm that beats the $\text{CMC}_{\max}$ bound of $\Omega(n^2 \log N)$. □

B Proof of Theorem 1

We first recall two lemmas from [4] used in our proof. The first lemma concerns the additive property of the cumulative complexity of two graphs [4, Lemma 3]; a consequence is that for any τ and DAG $\mathcal{G}$, the minimum cost to pebble τ instances of $\mathcal{G}$ is given by $\mathrm{CC}(\mathcal{G}^{\times\tau}) = \tau \cdot \mathrm{CC}(\mathcal{G})$. The second lemma concerns the predictor bound [4, Lemma 1], stating that, given a hint, a response from a random oracle can be successfully predicted with probability at most the size of the hint space divided by the size of the oracle's range.

Lemma 2 (Additive property of $\mathrm{CC}(\mathcal{G})$ [4]). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two node-disjoint DAGs and let $\mathcal{G}$ be their disjoint union. Then, $\mathrm{CC}(\mathcal{G}) = \mathrm{CC}(\mathcal{G}_1) + \mathrm{CC}(\mathcal{G}_2)$.*

Lemma 3 (Predictor bound [4]). *Let $B = b_1, ..., b_u$ be a sequence of random bits. Let $\mathcal{P}'$ be a randomized procedure that receives 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 that 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 Theorem 1 and present the proof.

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

$$\mathbf{Adv}^{(\tau,Q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \;\leq\; \frac{2q^2}{N} + t\,2^{-\gamma_{\max}} \;,$$

where $\gamma_{\max} := \frac{\tau \cdot \mathrm{CC}(\mathcal{G})\,[\log N - \log(q\tau)] - \mathrm{CMC}_{\max}(\mathcal{A})}{2t}$.

In [4], the proof in the single-instance hash-independent setting proceeds via two claims. The first states that a pebbling P, extracted from a transcript via the ex-post-facto pebbling extraction algorithm, is legal with high probability. The second states that, with high probability, the total number of pebbles in the extracted P is upper bounded by an essentially linear function of $\mathrm{CMC}_{\max}(\mathcal{A})$. 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 or not the construction is evaluated with a salt. 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 the graph as in the statement of the lemma. Let $\mathcal{A}$ be a q-query adversary in the (τ, Q)-sMH game with respect to C^{H}.

Transcript. An attack transcript T is a tuple $(\mathbf{q}_1, \ldots, \mathbf{q}_t, \sigma_{\mathcal{O}})$, where each $\mathbf{q}_i$ is a round of queries $(q_{i,1}, q_{i,2}, \ldots)$. Each $q_{i,j}$ is 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 $((v_{i,j}, \ell_{i,j}, \mathit{out}), \perp)$, where $v_{i,j}$ is a node, and $\ell_{i,j}$ is a node label. Here, $\sigma_{\mathcal{O}} = (\mathbf{x}, \mathbf{y})$ 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 algorithm of [4] to extract pebbling P of the graph $\mathcal{G}^{\times\tau}$ *even in a hash-dependent setting.*

Overview. Before extracting a pebbling, we determine, from T, a vector $\mathbf{x}^*$ of any τ inputs for which $\mathcal{A}$ correctly computed the output of C^{H}. As we consider adversaries that may output more values than they correctly compute, we extract the subset of inputs relevant to the pebbling. For each $k \in [\tau]$, the input $\mathbf{x}^*[k]$ defines a labeling of $\mathcal{G}$, and using these τ computations, we extract a pebbling of the graph $\mathcal{G}^{\times\tau}$. A pebbling $P = (P_0, \ldots, P_t)$ 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 is determined by 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 an input $\mathbf{x}^*[k]$, the node (v, k) will be added to the set P_i. Next, we keep only *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 by $H(x_{ij}) := y_{ij}$ if $(x_{ij}, y_{ij}) \in \mathbf{q}_i$ for some i, and otherwise $H(x) :=\perp$. We also set $H(\perp) :=\perp$.[7] As we are dealing with different labelings of the same graph, we need to extend the notation for labels and pre-labels. For each $k \in [\tau]$ 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 the node. Formally, a query (x, y) in the transcript is called correct (for some input $\mathbf{x}^*[k]$) if there exists a non-sink node v in $\mathcal{G}$ such that $x = \mathbf{pre\text{-}lab}_k(v)$, or there exists a sink node v in the graph such that $x = (v, \mathbf{lab}_k(v), \mathit{out})$. 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

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

$k \in [\tau]$ 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 $\mathcal{G}^{\times\tau}$ to a node in $\mathcal{G}$, so correctness is defined by the kth 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 each batch $\mathbf{q}_i$ in the order in which they appear in the transcript.

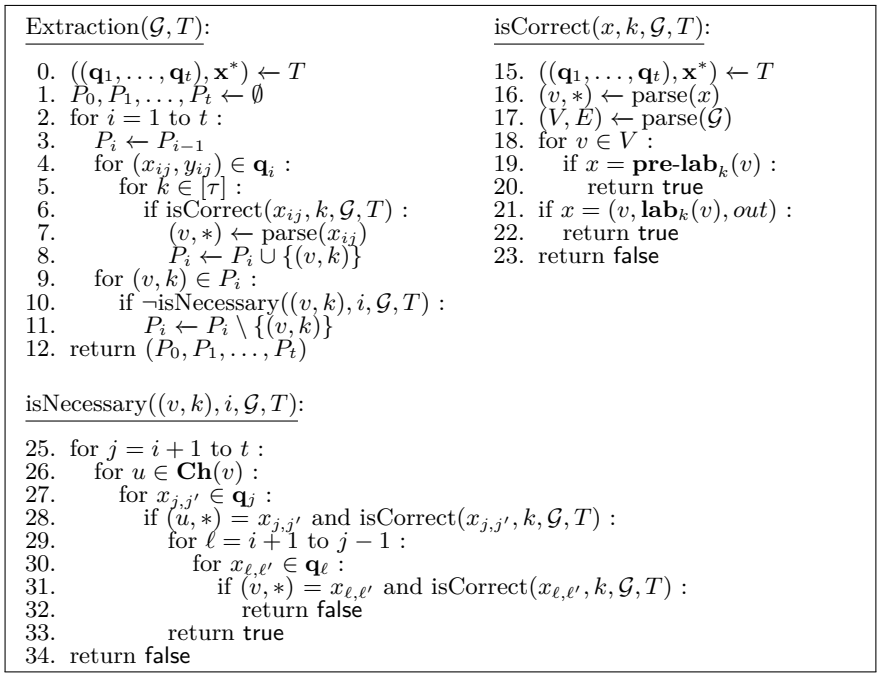
1. Add all nodes in P_{i-1} to P_i.
2. For each query $x_{i,j}$ in the current batch, if $x_{i,j}$ is a correct call for some node v and input $\mathbf{x}^*[k]$, add (v, k) to P_i.[8]
3. For each (v, k) in P_i that is not necessary, remove it from P_i.

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 those added by a correct oracle call. For the description of the ex-post-facto pebbling extraction in pseudo-code, see Fig. 5. With the pebbling extraction defined, we can analyze a pebbling extracted from a hash-dependent adversary. To this end, we prove three claims. To prove the first (legality), we use a prediction argument similar to that of [4], adapting the predictor to hash-dependent adversaries by extending its 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 $\log(q\tau)$ in the final bound. The final claim (collision-free transcript) follows from a standard collision argument.

Claim (Pebbling legality). For a fixed q-query pROM adversary $\mathcal{A}$, let T be a winning transcript corresponding to a q-query adversary $\mathcal{A}$ against (τ, Q)-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 [4] for a pebbling of $\mathcal{G}$ extracted from a hash-independent adversary computing a single instance. In such a setting, illegal pebble placement must correspond to an adversary predicting the output of 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 **pre-lab**(v), and thus the labels for

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



```
Extraction(G, T):
 0. ((q_1, ..., q_t), x*) ← T
 1. P_0, P_1, ..., P_t ← ∅
 2. for i = 1 to t :
 3.     P_i ← P_{i-1}
 4.     for (x_ij, y_ij) ∈ q_i :
 5.         for k ∈ [τ] :
 6.             if isCorrect(x_ij, k, G, T) :
 7.                 (v, *) ← parse(x_ij)
 8.                 P_i ← P_i ∪ {(v, k)}
 9.     for (v, k) ∈ P_i :
10.         if ¬isNecessary((v, k), i, G, T) :
11.             P_i ← P_i \ {(v, k)}
12. return (P_0, P_1, ..., P_t)

isCorrect(x, k, G, T):
15. ((q_1, ..., q_t), x*) ← T
16. (v, *) ← parse(x)
17. (V, E) ← parse(G)
18. for v ∈ V :
19.     if x = pre-lab_k(v) :
20.         return true
21. if x = (v, lab_k(v), out) :
22.     return true
23. return false

isNecessary((v, k), i, G, T):
25. for j = i + 1 to t :
26.     for u ∈ Ch(v) :
27.         for x_{j,j'} ∈ q_j :
28.             if (u, *) = x_{j,j'} and isCorrect(x_{j,j'}, k, G, T) :
29.                 for ℓ = i + 1 to j − 1 :
30.                     for x_{ℓ,ℓ'} ∈ q_ℓ :
31.                         if (v, *) = x_{ℓ,ℓ'} and isCorrect(x_{ℓ,ℓ'}, k, G, T) :
32.                             return false
33.                 return true
34. return false
```

Fig. 5: Algorithm for extracting a pebbling from an sMH attack transcript.

the parents of v.[9] 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 bounded 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 that 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 $k \in [\tau]$ to point to the input $\mathbf{x}^*[k]$ for which a label or pre-label is correct. Additionally, as P is a pebbling for $\mathcal{G}^{\times\tau}$, when adding pebbles, we translate the node indices (for queries labeling distinct instances of a construction) into node indices for (distinct subgraphs of) $\mathcal{G}^{\times\tau}$.

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 so that the predictor can still 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

[9] 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.

first correct call for u (if there is one). This extra index increases the hint space by a factor of q, giving rise to the q^2/N bound from the claim.

Proof (Pebbling legality). Let bad be the event that P is an illegal pebbling. That is, there is a 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 $\mathcal{G}^{\times\tau}$ 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 the label $\mathbf{lab}_k(u)$ is contained in $\mathbf{pre\text{-}lab}_k(v)$, though no *correct call* (w.r.t. 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 that takes a hint, depending on $\mathcal{A}$, from a space of size q^2. We then 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 from [4] 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 $\mathbf{x}^*$ only when it terminates. 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 otherwise malformed. $\mathcal{B}$ aims 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 $k \in [\tau]$. 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 and excluding 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 has not forwarded $\mathbf{pre\text{-}lab}_k(u)$ 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, 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 at least $1 - q^2/N$. □

Claim (Pebbling complexity). Let T be a winning transcript of a q-query t-time pROM adversary $\mathcal{A}$ against (τ, Q)-sMH. Let P be the pebbling extracted from the transcript. Then for any $\lambda \geq 0$:

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

over the choice of H and the coins of $\mathcal{A}$, where σ_i is the state (see Section 2.1).

Changes for multiple instances. An analogous result was proven in [4], 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 $\log(q\tau)$ 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, since it bounds the amount of memory required to store each pebbling set P_i, which is unaffected by hash-dependence.

As each node $(v, k) \in P_i$ is necessary, there is a later round containing 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 $\mathcal{B}$ that 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 (Pebbling complexity). 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 that 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 $q^{|P_i|}\tau^{|P_i|}2^{|\sigma_i|}$ 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 that a *k-critical call* for node $(v,k) \in P_i$ is 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 $J = \{(j_1,k_1),\ldots,(j_r,k_r)\} \in [q]^r \times [\tau]^r$ 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 $(v, l_1, \ldots, l_\delta)$,[10] $\mathcal{B}$ makes the following *recursive* check for correctness, starting by checking if, for any $k \in [\tau]$ 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 terminates at 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 $\mathbf{lab}_k(v)$; otherwise, 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 $\mathcal{A}$ has finished, $\mathcal{B}$ 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.

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

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

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

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

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

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

Claim (Collision-free transcript). Let $\mathcal{A}$ be a q-query (τ, Q)-sMH adversary, and bad be the event that $\mathcal{A}$ finds a collision between two labels $\mathbf{lab}_j(v)$ and $\mathbf{lab}_k(v)$ for some $j, k \in [\tau]$ and some node $v \in [n]$. Then $\Pr[\mathsf{bad}] \leq \frac{q^2}{N}$.

Proof (Collision-free transcript). To upper-bound the probability of bad, let $\mathbf{lab}_j(v)$ and $\mathbf{lab}_k(v)$ be the labels for a node v. A collision in node labels corresponds to a collision between 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

$$\Pr[\mathsf{bad}] \leq \frac{1}{N}\sum_1^n q_i^2 \leq \frac{1}{N}\left(\sum_1^n q_i\right)^2 \leq \frac{q^2}{N}\,. \qquad \square$$

Now let $\mathcal{W}$ be the event that $\mathcal{A}$ wins the (τ, Q)-sMH$_{\mathsf{C}^\mathsf{H}}$ game and bad be the collision event defined above. We have

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

For any q-query t-time adversary $\mathcal{A}$ in the (τ, Q)-sMH game w.r.t. C^H that produces a collision-free transcript T, let P be the ex-post-facto pebbling of $\mathcal{G}^{\times\tau}$ extracted from T.

Let $\mathcal{C}_\lambda$ be the event that P is legal and $\forall i : |P_i| \leq (|\sigma_i|+\lambda)/(\log N - \log(q\tau))$. Then, by applying the pebbling legality and pebbling complexity claims above, we have $\Pr[\neg\mathcal{C}_\lambda] < q^2/N + t2^{-\lambda}$, i.e., $\Pr[\mathcal{C}_\lambda] > 1 - q^2/N - t2^{-\lambda}$.

Now, if $\Pr[\mathcal{W} \mid \neg\mathsf{bad}] > q^2/N + t\,2^{-\lambda}$, then some winning run must lie in $\mathcal{C}_\lambda$, and thus

$$\sum_i |P_i^\star| \ \leq\ \frac{\sum_i |\sigma_i^\star| + t\lambda}{\log N - \log(q\tau)} \ \leq\ \frac{\mathrm{CMC}_{\max}(\mathcal{A}) + t\lambda}{\log N - \log(q\tau)}\ ,$$

where $P_i^\star$ and $\sigma_i^\star$ denote the pebbling sets and states in this particular winning run. The second inequality crucially relies on the *maximum* CMC metric.

Note that since we are considering a winning run, the pebbling $P^\star$ is complete. Moreover, since there are no collisions, the pebbled graphs are disjoint, and hence by the definition and the additive property of pebbling complexity (Lemma 2), we have $\sum_i |P_i^\star| \geq \mathrm{CC}(\mathcal{G}^{\times\tau}) = \tau\mathrm{CC}(\mathcal{G})$. Combining this with the above equation, we get

$$\tau\mathrm{CC}(\mathcal{G}) \leq \frac{\mathrm{CMC}_{\max}(\mathcal{A}) + t\lambda}{\log N - \log(q\tau)} \ ,$$

or equivalently

$$\lambda \geq \frac{\tau\,\mathrm{CC}(\mathcal{G})[\log N - \log(q\tau)] - \mathrm{CMC}_{\max}(\mathcal{A})}{t} =: 2\gamma_{\max} \ .$$

In other words, if $\lambda < 2\gamma_{\max}$, then $\Pr[\mathcal{W} \mid \neg\mathsf{bad}] \leq q^2/N + t\,2^{-\lambda}$.

Putting these terms together, we get

$$\mathbf{Adv}^{(\tau,Q)\text{-sMH}}_{\mathsf{C}^{\mathsf{H}}}(\mathcal{A}) \leq \Pr[\mathsf{bad}] + \Pr[\mathcal{W}|\neg\mathsf{bad}] \leq \frac{2q^2}{N} + t2^{-\lambda} \ .$$

We set $\lambda := \gamma_{\max}$, which is less than $2\gamma_{\max}$ if $\gamma_{\max}$ is positive, and obtain the bound in the theorem statement. If $\gamma_{\max}$ is not positive, the theorem holds trivially. □

C Setting P

P is chosen to minimize $\mathbf{Adv}^{\text{SA-Guess}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + t\,2^{-\gamma_{\max}} + \frac{(nm\ell+4q)^2}{N} + \frac{m\ell}{K}$. Since only the first two terms depend on P, it is sufficient to choose P to minimize the sum $\mathbf{Adv}^{\text{SA-Guess}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B}) + t\,2^{-\gamma_{\max}}$. Note that $\mathbf{Adv}^{\text{SA-Guess}}_{\mathcal{P}_m,\ell,\mathsf{Gen}}(\mathcal{B})$ is non-decreasing in P, as having more queries can only increase $\mathcal{B}$'s advantage, and $t\,2^{-\gamma_{\max}}$ is decreasing in P, since the larger P is, the larger $\gamma_{\max}$ is by definition. The minimum of 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 at which the adversary can no longer 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 $(1/N)^{m-c} \leq \mathbf{Adv}^{\text{SA-Guess}}_{\mathcal{P}^{\mathsf{H}}_m,\ell,\mathsf{Gen}}(\mathcal{B}) \leq 1$.

Let $\Delta = \mathrm{CMC}_{\max}(\mathcal{P}_m) + mt\log N$. We first lower bound P by finding the smallest P such that $t\,2^{-\gamma_{\max}} < 1$. This occurs when $\gamma_{\max} > \log t$, i.e., $(P+1)\mathrm{CC}(\mathcal{G})(\log N - \log(q(P+1))) - (\mathrm{CMC}_{\max}(\mathcal{A}) + \Delta) > 2t\log t$. Rearranging gives

$$\begin{aligned}\mathrm{CMC}_{\max}(\mathcal{A}) + \Delta + 2t\log t &< (P+1)\mathrm{CC}(\mathcal{G})(\log N - \log(q(P+1))) \\ &< (P+1)\mathrm{CC}(\mathcal{G})\log N \ ,\end{aligned}$$

which, after expanding Δ gives

$$P > \frac{\mathrm{CMC}_{\max}(\mathcal{A}) + \mathrm{CMC}_{\max}(\mathcal{P}_m) + mt\log N + 2t\log t}{\mathrm{CC}(\mathcal{G})\log N} - 1 \ . \tag{1}$$

We then upper bound P by finding the largest P such that $t\,2^{-\gamma_{\max}} > (\frac{1}{N})^{m-c}$. This occurs when $\gamma_{\max} < (m-c)\log N + \log t$, i.e., $(P+1)\mathrm{CC}(\mathcal{G})(\log N - \log(q(P+1))) - (\mathrm{CMC}_{\max}(\mathcal{A}) + \Delta) < 2t((m-c)\log N + \log t)$, which in turn rearranges to give

$$(P+1)(\log N - \log(q(P+1))) < \frac{\mathrm{CMC}_{\max}(\mathcal{A}) + \Delta + 2t((m-c)\log N + \log t)}{\mathrm{CC}(\mathcal{G})} \ .$$

We can assume that $q(P+1) < \sqrt{N}$ (otherwise the collision bounds become dominant), and this gives:

$$\begin{aligned}\frac{\mathrm{CMC}_{\max}(\mathcal{A}) + \Delta + 2t((m-c)\log N + \log t)}{\mathrm{CC}(\mathcal{G}))} &> (P+1)(\log N - \log(\sqrt{N})) \\ &> (P+1)\log(\sqrt{N}) \ ,\end{aligned}$$

which, after expanding Δ and a final rearrangement gives

$$P < 2\ \frac{\mathrm{CMC}_{\max}(\mathcal{A}) + \mathrm{CMC}_{\max}(\mathcal{P}_m) + (3tm - 2tc)\log N + 2t\log t}{\mathrm{CC}(\mathcal{G})\log N} - 1 \ . \tag{2}$$

Noting that the two bounds in Eqs. (1) and (2) are a constant factor apart, we can simplify P asymptotically as

$$P = \Theta\left(\frac{\mathrm{CMC}_{\max}(\mathcal{A}) + \mathrm{CMC}_{\max}(\mathcal{P}_m) + mt\log N + t\log t}{\mathrm{CC}(\mathcal{G})\log N}\right) \ .$$

In other words, we set P to be of the order of the number of complete chains that the adversary can compute given a CMC allowance. The $\mathrm{CMC}_{\max}(\mathcal{A})$ term is expected to dominate the numerator, and hence the optimal value of P will be $\Theta\left(\frac{\mathrm{CMC}_{\max}(\mathcal{A})}{\mathrm{CC}(\mathcal{G})\log N}\right)$.