Memory-Hard Puzzles in the Standard Model with Applications to Memory-Hard Functions and Resource-Bounded Locally Decodable Codes

Mohammad Hassan Ameri* Alexander R. Block* Jeremiah Blocki*

September 21, 2021

Abstract

We formally introduce, define, and construct *memory-hard puzzles*. Intuitively, for a difficulty parameter t, a cryptographic puzzle is memory-hard if any parallel random access machine (PRAM) algorithm with "small" (amortized) area-time complexity ($\ll t^2$) cannot solve the puzzle; moreover, such puzzles should be both "easy" to generate and be solvable by a sequential RAM algorithm running in time t. Our definitions and constructions of memory-hard puzzles are in the standard model, assuming the existence of indistinguishability obfuscation ($i\mathcal{O}$) and one-way functions (OWFs), and additionally assuming the existence of a *memory-hard language*. Intuitively, a language is memory-hard if it is undecidable by any PRAM algorithm with "small" (amortized) area-time complexity, while a sequential RAM algorithm running in time t can decide the language. Our definitions and constructions of memory-hard objects are the first such definitions and constructions in the standard model without relying on idealized assumptions (such as random oracles).

We give two applications which highlight the utility of memory-hard puzzles. For our first application, we give a construction of a (one-time) *memory-hard function* (MHF) in the standard model, using memory-hard puzzles and additionally assuming $i\mathcal{O}$ and OWFs. For our second application, we show any cryptographic puzzle (e.g., memory-hard, time-lock) can be used to construct *resource-bounded locally decodable codes* (LDCs) in the standard model, answering an open question of Blocki, Kulkarni, and Zhou (ITC 2020). Resource-bounded LDCs achieve better rate and locality than their classical counterparts under the assumption that the adversarial channel is resource bounded e.g., a low-depth circuit. Prior constructions of MHFs and resource-bounded LDCs required idealized primitives like random oracles.

1 Introduction

Memory-Hardness is an important notion in the field of cryptography that is used to design egalitarian proofs of work and to protect low entropy secrets (e.g., passwords) against brute-force attacks. Over the last decade, there has been a rich line of both theoretical and applied work in constructing and analyzing Memory-Hard Functions [FLW14, AS15, BDK16, AB16, ABP17, ACP+17, AT17, BZ17, ABH17, ABP18, BHK+19, CT19]. Ideally, one wants to prove that any algorithm evaluating the function (possibly on multiple distinct inputs) has high (amortized) Area-Time complexity (aAT) [AS15] (asymptotically equivalent to the notions of (amortized) Space-Time complexity and cumulative memory complexity (cmc) in the literature). Currently, security proofs for memory-hard objects rely on idealized assumptions such as the existence of random oracles [AS15, ACP+17, AT17, ABP18] or other ideal objects such as ideal ciphers or permutations [CT19]. Informally, a function f is memory-hard if there is a sequential algorithm computing f in time t, but any parallel algorithm computing f (possibly

*Purdue University. Email: {mameriek,block9,jblocki}@purdue.edu

on multiple distinct inputs) has high aAT complexity, e.g., $t^{2-\varepsilon}$ for small constant $\varepsilon > 0$. An important open question is to construct provably secure memory-hard objects in the standard model.

In this work we focus specifically on memory-hard puzzles. Cryptographic puzzles are cryptographic primitives that have two desirable properties: (1) for a target solution s, it should be "easy" to generate a puzzle Z with solution s; and (2) solving the puzzle Z to obtain solution s should be "difficult" for any algorithm $\mathcal{A}$ with "insufficient resources". Such puzzles have seen a wide range of applications, including use in cryptocurrency, handling junk mail, and constructing time-released encryption schemes [DN93, RSW96, JJ99, Nak]. For example, the well-known and studied notion of *time-lock puzzles* [RSW96, BN00, GMPY11, MMV11, BGJ+16, MT19] requires that for difficulty parameter t and security parameter λ, a sequential (i.e., non-parallel) machine can generate a puzzle in time $\mathrm{poly}(\lambda, \log(t))$ and solve the puzzle in time $t \cdot \mathrm{poly}(\lambda)$, but requires that any parallel algorithm running in sequential time significantly less than t (i.e., any polynomial size circuit of depth smaller than t) cannot solve the puzzle, except with negligible probability (in the security parameter). In the context of memory-hard puzzles we want to ensure that the puzzles are easy to generate, but that any algorithm solving the puzzle has high aAT complexity. More concretely, we require that the puzzles can be generated (resp. solved) in time $\mathrm{poly}(\lambda, \log(t))$ (resp. $t \cdot \mathrm{poly}(\lambda)$) on a sequential machine while any algorithm solving the puzzle has aAT complexity at most $t^{2-\varepsilon}$ for some small constant $\varepsilon > 0$. We remark that any sequential machine solving the puzzle in time at most $t \cdot \mathrm{poly}(\lambda)$ will have aAT complexity at most $t^2 \cdot \mathrm{poly}(\lambda)$ so a lower bound of $t^{2-\varepsilon}$ for the aAT complexity of our puzzles would be nearly tight.

In this work, we ask the following questions:

Is it possible to construct memory-hard puzzles under standard cryptographic assumptions? If yes, what applications of memory-hard puzzles can we find?

1.1 Our Results

We formally introduce and define the notion of *memory-hard puzzles*. Inspired by time-lock puzzles and memory-hard functions, we define memory-hard puzzles *without* idealized assumptions. Intuitively, we say that a cryptographic puzzle is memory-hard if any parallel random access machine (PRAM) algorithm with "small" aAT complexity cannot solve the puzzles. This is in contrast with time-lock puzzles, which require that any algorithm running in "small" sequential time (i.e., any low-depth circuit) cannot solve the puzzle. For both memory-hard and time-lock puzzles, the puzzles should be "easy" to generate; i.e., in sequential time $\mathrm{poly}(\lambda, \log(t))$.

Similar to the construction of time-lock puzzles of Bitansky et al. [BGJ+16], we construct memory-hard puzzles assuming the existence of a suitable *succinct randomized encoding scheme* [IK00, AIK04, BGL+15, BGJ+16, LPST16, App17, GS18], and the additional assumption that there exists a language which is "suitably" memory-hard. Towards this end, we formally introduce and define *memory-hard languages*: such languages, informally, require that (1) the language is decidable by a family of *uniformly succinct circuits*—succinct circuits which are computable by a uniform algorithm—of appropriate size; and (2) any PRAM algorithm deciding the language must have "large" aAT complexity. Our constructions are primarily of theoretical interest, as known constructions of randomized encodings rely on expensive primitives such as indistinguishability obfuscation ($i\mathcal{O}$) [BGI+01, GGH+13, KLW15, LM18, BIJ+20, AP20, BDGM20, JLS21]; we make no claims about the practical efficiency of our constructions.

Theorem 1.1 (Informal, see Theorem 4.9)**.** *Assuming the existence of succinct randomized encodings and a memory-hard language, there exists a construction of a memory-hard puzzle.*

We stress that our construction does not rely on an explicit instance of a memory-hard language: the existence of such a language suffices to prove memory-hardness of the constructed puzzle (this mirrors the construction of [BGJ+16]). We use succinct randomized encoding scheme of Garg and Srinivasan

[GS18], which is instantiated from $i\mathcal{O}$ for circuits and somewhere statistically binding hash functions [HW15, KLW15, OPWW15].[1]

It is important to note that even if we defined memory-hard puzzles in an idealized model (e.g., the random oracle model), memory-hard functions do not directly yield memory-hard puzzles. Cryptographic puzzles stipulate that for parameters t, λ the puzzle generation algorithm needs to run in time $\text{poly}(\lambda, \log(t))$. However, using a memory-hard function to generate a cryptographic puzzle would require the generation algorithm to compute the memory-hard function, which would yield a generation algorithm running in time (roughly) proportional to $t \cdot \text{poly}(\lambda)$.

1.1.1 Application 1: Memory-Hard Functions

We demonstrate the power of memory-hard puzzles via two applications. For our first application, we use memory-hard puzzles to construct a (one-time secure) *memory-hard function* (MHF) in the standard model. As part of this construction, we formally define (one-time) memory-hard functions in the standard model, without idealized primitives. We emphasize that all prior constructions of memory-hard functions rely on idealized primitives such as random oracles [FLW14, AS15, BDK16, AB16, ABP17, ACP+17, AT17, BZ17, ABH17, ABP18, BHK+19] or ideal ciphers and permutations [CT19]. In fact, prior definitions of memory-hardness were with respect to an idealized model such as the parallel random oracle model, e.g., [AS15].

Recall that a function f is memory-hard if it can be computed by a sequential machine in time t (and thus uses space at most t), but any PRAM algorithm evaluating f (possibly on multiple distinct inputs) has high amortized area-time (aAT) complexity, e.g., at least $t^{2-\varepsilon}$ for small constant $\varepsilon > 0$. One-time security stipulates that any attacker with low aAT complexity cannot distinguish between $(x, f(x))$ and (x, r) with non-negligible advantage when r is a uniformly random bit string.[2] Assuming the existence of indistinguishability obfuscation, puncturable pseudo-random functions, and memory-hard puzzles, we give a construction of one-time secure memory-hard functions.

Theorem 1.2 (Informal, see Theorem 5.4)**.** *Assuming the existence of indistinguishability obfuscation, a puncturable pseudo-random function family, and memory-hard puzzles, there exists a construction of a one-time secure memory-hard function.*

We stress that, to the best of our knowledge, this is the *first* construction of a memory-hard function under standard cryptographic assumptions and the additional assumption that a memory-hard puzzle exists. Given Theorem 1.1, we construct memory-hard functions from standard cryptographic assumptions only additionally assuming the existence of a memory-hard language. As stated previously, all prior constructions of memory-hard functions were proven secure under idealized assumptions, such as the random oracle model or ideal cipher and permutation models.

We also conjecture that our scheme is multi-time secure as well: if an attacker with low aAT complexity, say some g, cannot distinguish between $(x, f(x))$ and (x, r) for uniformly random bit string r, then an attacker with aAT complexity at most $m \cdot g$ cannot distinguish between $(x_i, f(x_i))$ and (x_i, r_i) for m distinct inputs $x_1, \ldots, x_m$ and uniformly random strings $r_1, \ldots, r_m$. However, we are unable to formally prove this due to some technical barriers in the security proof. At a high level, this is due to the fact that allowing the attacker to have higher aAT complexity (e.g., $m \cdot g$) eventually leads to an attacker with large enough aAT complexity to simply solve the underlying memory-hard

[1] Such hash functions can generate hashing key that statistically binds the i-th input bit. For example, a hash output y may have many different preimages, but all preimages have the same i-th bit. Construction of such hash functions exist under standard cryptographic assumptions such as DDH and LWE, among others [OPWW15].

[2] Our one-time security definition differs from those in prior literature (e.g., [AS15, ACP+17]), and is in fact stronger than requiring that an adversary with insufficient resourced cannot compute the MHF. However, we remark that in the random oracle model, for random oracle H, any MHF f immediately yields a function $f'(x) = H(f(x))$ which is indistinguishable from random to any adversary that cannot compute $f(x)$.

puzzle that is used in the MHF construction, thus allowing the adversary to distinguish instances of the MHF instance. See Section 5.1 for details.

1.1.2 Application 2: Locally Decodable Codes for Resource Bounded Channels

As our second application, we use cryptographic puzzles to obtain efficient *locally decodable codes for resource-bounded channels* [BKZ20]. A (q, δ, p)-locally decodable code (LDC) $C[K, k]$ over some alphabet Σ is an error-correcting code with encoding function $\mathsf{Enc}\colon \Sigma^k \to \{0,1\}^K$ and probabilistic decoding function $\mathsf{Dec}\colon \{1, \ldots, k\} \to \Sigma$ satisfying the following properties. For any message x, the decoder, when given oracle access to some $\tilde{y}$ such that $\Delta(\tilde{y}, \mathsf{Enc}(x)) \leqslant \delta K$, makes at most q queries to its oracle and outputs x_i with probability at least p, where Δ is the Hamming distance. The *rate* of the code is k/K, the *locality* of the code is q, the *error tolerance* is δ, and the *success probability* is p. Classically (i.e., the adversarial channel introducing errors is computationally unbounded), there is an undesirable trade-off between the rate k/K and locality, e.g., if $q = \mathrm{polylog}(k)$ then $K \gg k$.

Modeling the adversarial channel as computationally unbounded, as in the classical setting, may be overly pessimistic. Moreover, it has been argued that any real world communication channel can be reasonably modeled as a resource-bounded channel [Lip94, BKZ20]. A *resource-bounded channel* is an adversarial channel that is assumed to have some constrained resource (e.g., the channel is a low-depth circuit), and a resource-bounded LDC is a LDC that is secure with respect to some class of resource-bounded channels $\mathbb{C}$. Arguably, error patterns (even random ones) encountered in nature can be modeled by some (not necessarily known) resource-bounded algorithm which simulates the same error pattern, and thus these channels are well-motivated by real world channels. For example, sending a message from Earth to Mars takes between (roughly) 3 and 22 minutes when traveling at the speed of light; this limits the depth of any computation that could be completed before the (corrupted) codeword is delivered. Furthermore, examining LDCs resilient against several resource-bounded channels has led to better trade-offs between the rate and locality than their classical counterparts [Lip94, MPSW05, BGH+06, GS16, SS16, BGGZ19].

Recently, Blocki, Kulkarni, and Zhou [BKZ20] construct LDCs for resource-bounded channels with locality $q = \mathrm{polylog}(k)$ and constant rate $k/K = \Theta(1)$, but their construction relies on random oracles. We use cryptographic puzzles to modify the construction of [BKZ20] to obtain resource-bounded LDCs without random oracles. Given any cryptographic puzzle that is secure against some class of adversaries $\mathbb{C}$, we construct a locally decodable code for Hamming errors that is secure against the class $\mathbb{C}$ resolving an open problem of Blocki, Kulkarni, and Zhou [BKZ20].

Theorem 1.3 (Informal, see Theorem 6.8)**.** *Let $\mathbb{C}$ be a class of algorithms such that there exists a cryptographic puzzle that is unsolvable by adversaries in this class. There exists a construction of a locally decodable code for Hamming errors with constant rate and poly-logarithmic locality that is secure against the class $\mathbb{C}$.*

We can instantiate our LDC with any (concretely secure) cryptographic puzzle. In particular, the time-lock puzzles of Bitansky et al. [BGJ+16] directly give us LDCs secure against small-depth channels, and our memory-hard puzzle construction gives us LDCs secure against any channel with low aAT complexity. Our LDC construction for resource bounded Hamming channels can also be extended to resource-bounded insertion-deletion (InsDel) channels by leveraging recent "Hamming-to-InsDel" LDC compilers [OPC15, BBG+20, BB21].

Theorem 1.4 (Informal, see Corollary 6.10)**.** *Let $\mathbb{C}$ be a class of algorithms such that there exists a cryptographic puzzle that is unsolvable by adversaries in this class. There exists a construction of a locally decodable code for insertion-deletion errors with constant rate and poly-logarithmic locality that is secure against the class $\mathbb{C}$.*

1.1.3 Challenges in Defining Memory-Hardness

Defining the correct machine model and cost metric for memory-hard puzzles is surprisingly difficult. As PRAM algorithms and aAT complexity are used extensively in the study of MHFs, it is natural to use the same machine model and cost metric. However, aAT complexity introduced subtleties in the analysis of our memory-hard puzzle construction: like [BGJ+16], we rely on parallel amplification in order to construct an adversary which breaks our memory-hard language assumption. Whereas parallel amplification does not significantly increase the depth of a computation (which is the metric used by [BGJ+16]), any amplification directly increases the aAT complexity of an algorithm by a factor proportional to the number of amplification procedures performed. This requires careful consideration in our security reductions.

One may also attempt to define memory-hard languages as languages with aAT complexity $t^{2-\varepsilon}$, for small constant $\varepsilon > 0$, that are also decidable by single-tape Turing machines (à la [BGJ+16]) in time t, rather than by uniformly succinct circuit families. However, we demonstrate a major hurdle towards this definition. In particular, we show that any single-tape Turing machine running in time t can be simulated by any PRAM algorithm with aAT complexity at least $\Omega(t^{1.8} \cdot \log(t))$ (see Section 8). Taking this approach we could not hope obtain memory hard puzzles with aAT complexity $t^{2-\varepsilon}$ as we can rule out the existence of memory-hard languages with aAT $\gg t^{1.8}$. To contrast, under our uniformly succinct definition, we can provide a concrete candidate language with aAT complexity plausibly as high as $t^{2-\varepsilon}$ such that the language is also decidable by a uniformly succinct circuit family of size $\tilde{O}(t)$.[3] Furthermore, we show that our definition is essentially minimal, i.e., we can use memory-hard puzzles to construct memory-hard languages under the modest assumption that the puzzle solving algorithm is uniformly succinct (see Section 4.2).

1.2 Prior Work

Cryptographic puzzles are functions which require some specified amount of resources (e.g., time or space) to compute. Time-lock puzzles, introduced by Rivest, Shamir, and Wagner [RSW96] extending the study of timed-released cryptography of May [May], are puzzles which require large sequential time to solve: any circuit solving the puzzle has large depth. [RSW96] proposed a candidate time-lock puzzle based on the conjectured sequential hardness of exponentiation in RSA groups, and the proposed schemes of [BN00, GMPY11] are variants of this scheme. Mahmoody, Moran, and Vadhan [MMV11] give a construction of *weak* time-lock puzzles in the random oracle model, where "weak" says that both a puzzle generator and puzzle solver require (roughly) the same amount of computation, whereas the standard definition of puzzles requires the puzzle generation algorithm to be much more efficient than the solving algorithm. Closer to our work, Bitansky et al. [BGJ+16] construct time-lock puzzles using succinct randomized encodings, which can be instantiated from one-way functions, indistinguishability obfuscation, and other assumptions [GS18]. Recently, Malavolta and Thyagarajan [MT19] introduce and construct homomorphic time-lock puzzles: puzzles where one can compute functions over puzzles without solving them. We remark that continued exploration of indistinguishability obfuscation has pushed it closer and closer to being instantiated from well-founded cryptographic assumptions such as learning with errors [JLS21].

Memory-hard functions (MHFs), introduced by Percival [Per09], have enjoyed rich lines of both theoretical and applied research in construction and analysis of these functions [CT19, AS15, AT17, BDK16, FLW14, AB16, ABP17, ABP18, ACP+17, BZ17, ABH17, BHK+19]. The security proofs of all prior MHF candidates rely on idealized assumptions (e.g., random oracles [AS15, ACP+17, AT17, ABP18, BRZ18]) or other ideal objects (e.g., ideal ciphers or permutations [CT19]). The notion of *data-independent* MHFs—MHFs where the data-access pattern of computing the function, say, via

[3] In fact, one can provably show that the aAT complexity is $t^{2-\varepsilon}$ in the random oracle model.

a RAM program, is independent of the input—has also been widely explored. Data-independent MHFs are attractive as they provide natural resistance to side-channel attacks. However, building data-independent memory-hard functions (iMHFs) comes at a cost: *any* iMHF has amortized space-time complexity at most $O(N^2 \cdot \mathrm{loglog}(N)/\log(N))$ [AB16], while data-dependent MHFs were proved to have maximal complexity $\Omega(N^2)$ in the parallel random oracle model [ACP+17] (here, N denotes the run time of the honest sequential evaluation algorithm). Recently, Ameri, Blocki, and Zhou [ABZ20] introduced the notion of *computationally data-independent* memory-hard functions: functions which appear data-independent to a computationally bounded adversaries. This relaxation of data-independence allowed [ABZ20] to circumvent known barriers in the construction of data-independent MHFs as long as certain assumptions on the tiered memory architecture (RAM/cache) hold.

LDC constructions, like all code constructions, generally follow one of two channel models: the Hamming channel where worst-case bit-flip error patterns are introduced, and the Shannon channel where symbols are corrupted by an independent probabilistic process. Probabilistic channels may be too weak to capture natural phenomenon, while Hamming channels often limit the code constructions we are able to obtain. In the case of the Hamming channel, the channel is assumed to be information theoretic; i.e., it has unbounded computational time and power. Protecting against such unbounded errors is desirable but often has undesirable trade-offs. For example, current constructions of locally decodable codes with efficient (i.e., poly-time) encodings an obtain any constant rate $R < 1$, are robust to $\delta < (1-R)$-fraction of errors, but have query complexity $2^{O(\sqrt{\log n \log\log n})}$ for codeword length n [KMRS17]. If one instead focuses on obtaining low query complexity, one can obtain schemes with codewords of length sub-exponential in the message size while using a constant number $q \geqslant 3$ queries [Yek08, DGY11, Efr12].

These undesirable trade-offs have lead to a long line of work examining LDCs (and codes in general) with relaxed assumptions [Lip94, MPSW05, BGH+06, GS16, SS16, BGGZ19]. Two relaxations closely related to our work are due to Ostrovsky, Pandey, and Sahai [OPS07] and Blocki, Kulkarni, and Zhou [BKZ20]. [OPS07] introduce and construct *private* Hamming LDCs: locally decodable codes in the secret key setting, where the encoder and decoder share a secret key that is unknown to the (unbounded) channel. [BKZ20] analyze Hamming LDCs in the context of *resource-bounded channels*. The LDC construction of [BKZ20] bootstraps off of the private Hamming LDC construction of [OPS07], obtaining Hamming LDCs in the random oracle model assuming the existence of functions which are uncomputable by the channel.

While Hamming LDCs have enjoyed decades of research [KT00, STV99, DGY10, Efr09, KW03, KMRZS17, KS16, Yek08, Yek12], the study of *insertion-deletion* LDCs (or InsDel LDCs) remains scarce. An InsDel LDC is a locally decodable code that is resilient to bounded adversarial insertion-deletion errors. In the non-locally decodable setting, there has been a rich line of research into insertion-deletion codes [Lev66, KLM04, GW17, HS17, GL19, GL18, HSS18, HS18, BGZ18, CJLW18, CHL+19, CJLW19, HRS19, Hae19, SB19, CGHL20, CL20, GHS20, LTX20], and only recently have efficient InsDel codes with asymptotically good information rate and error tolerance been well-understood [HS18, Hae19, HRS19, GHS20, LTX20]. Ostrovsky and Paskin-Cherniavsky [OPC15] and Block et al. [BBG+20] give a compiler which transforms any Hamming LDC into an InsDel LDC with a polylogarithmic blow-up in the locality. Block and Blocki [BB21] recently extended the compiler of [BBG+20] to the private and resource-bounded settings.

2 Technical Overview

Our construction of memory-hard puzzles relies on two key technical ingredients. First we require the existence of a language $\mathcal{L} \subseteq \{0,1\}^*$ that is suitably *memory-hard*. Given such a language, we additionally require *succinct randomized encodings* [BGL+15, LPST16, GS18] for succinct circuits. With these two objects, we construct *memory-hard puzzles*. We discuss the key ideas behind memory-hard languages and puzzles.

2.1 Memory-Hard Languages

Our definition of memory-hard languages is inspired by the notion of non-parallelizing languages,[4] which are required by Bitansky et al. [BGJ+16] to construct time-lock puzzles (also using succinct randomized encodings). Informally, we say a language $\mathcal{L}$ is *memory-hard* if it satisfies the following two conditions. For function t and small constant $\varepsilon > 0$:

1. For every $\lambda \in \mathbb{N}$, the language $\mathcal{L}_\lambda := \mathcal{L} \cap \{0,1\}^\lambda$ is decidable by a *uniformly succinct circuit* $C_{t,\lambda}$ of size $t \cdot \text{polylog}(t)$ for $t := t(\lambda)$. Informally, a circuit family $\{C_{t,\lambda}\}_{\lambda\in\mathbb{N}}$ is *succinctly describable* [BGT14, GS18] if there exists a smaller circuit family $\{C'_{t,\lambda}\}_{\lambda\in\mathbb{N}}$ such that for every $n \in \mathbb{N}$, $|C'_n| < |C_n|$ and on input gate number g of $C_{t,\lambda}$, we have that $C'_{t,\lambda}(g)$ outputs the indices of the input gates of g and the function f_g computed by gate g. Uniformly succinct requires that there exists a sequential algorithm running in time $\text{polylog}(t, \lambda)$ that outputs the description of the succinct circuit $C'_{t,\lambda}$ for every λ.
2. Any family of PRAM algorithms deciding $\mathcal{L}_\lambda$ for every λ has large area-time complexity; i.e., at least $t^{2-\varepsilon}$.

Our definition of memory-hard languages is essentially minimal, as one can construct memory-hard languages from memory-hard puzzles under the modest assumption that the puzzle solving algorithm is uniformly succinct; see Section 4.2 for details.

We complement our definition of memory-hard languages by providing a concrete construction of a candidate memory-hard language. We define a language $\mathcal{L}_\lambda = \mathcal{L} \cap \{0,1\}^\lambda$ that is decidable by a uniformly succinct circuit $C_{t,\lambda}$ of size $t \cdot \text{polylog}(t)$ and provably has large area-time complexity at least $t^2/\text{polylog}(t)$ in the random oracle model. This language relies on a memory-hard function based on the so-called the *powers of two graph* from folklore. This graph has $N = 2^n$ nodes with directed edges of the form $(u, u + 2^i)$ for each $i \geqslant 0$ such that $u + 2^i \leqslant N$ and satisfies a combinatial property called depth-robustness which is sufficient for constructing memory hard functions in the parallel random oracle model [ABP17]. Crucially, the description of the powers of two graph is sufficiently simple that it can be fully encoded by a uniformly succinct circuit. We remark that other randomized constructions of depth-robust graphs such the one used in the DRSample memory-hard function [ABH17] cannot be used to construct memory-hard languages as the graphs are not uniformly succinct. See Section 7 for more discussion. We emphasize that we only know how to prove our candidate language is memory-hard in the random oracle model, but that it is very plausible that our defined language will remain memory-hard for certain concrete instantiations of the random oracle (e.g., SHA3). While we conjecture that our constructed language is memory-hard in the standard model, proving this conjecture would require major advances in circuit lower bounds.

On the negative side, if we require our memory-hard language to be decidable by a single-tape Turing machine in time $t = t(\lambda)$, then the language is only secure against PRAM algorithms with aAT complexity $o(t \cdot \log(t))$. We show this by proving that any single-tape Turing machine running in time $t = t(\lambda)$ for λ-bit inputs can be simulated by a PRAM algorithm in time $O(t)$ using with space at most $t^{0.8} \cdot \log(t)$. As aAT complexity is upper bounded by the maximum space of a computation times the maximum time of a computation, this gives us our desired bound. See Section 8 for more details.

[4] Informally, a language is non-parallelizing if any polynomial sized circuit deciding the language has large depth.

2.2 Memory-Hard Puzzles

We construct memory-hard puzzles by using succinct randomized encodings for succinct circuits and additionally assuming that a (suitably) memory-hard language exists. Informally, a succinct randomized encoding for succinct circuits consists of two algorithms $\mathsf{sRE.Enc}$ and $\mathsf{sRE.Dec}$ where $\widehat{C}_{x,G} \leftarrow \mathsf{sRE.Enc}(1^\lambda, C', x, G)$ takes as input a security parameter λ, a succinct circuit C' describing a larger circuit C with G gates and an input $x \in \{0,1\}^*$ and outputs a randomized encoding $\widehat{C}$ in time $\mathrm{poly}(|C'|, \lambda, \log(G), |x|)$. The decoding algorithm $\mathsf{sRE.Dec}(\widehat{C}_{x,G})$ outputs $C(x)$ in time at most $G \cdot \mathrm{poly}(\log(G), \lambda)$. Note that the running time requirement ensures $\mathsf{sRE.Enc}$ cannot simply compute $C(x)$. Intuitively, security implies that the encoding $\widehat{C}_{x,G}$ reveals nothing more than $C(x)$ to a computationally bounded attacker.

We extend ideas from [BGJ+16] to construct memory-hard puzzles from succinct randomized encodings. In particular, the generation algorithm $\mathsf{Puz.Gen}(1^\lambda, t, s)$ first constructs a Turing machine $M_{s,t}$ that on any input runs for t steps then outputs s, where $t = t(\lambda)$ and $s \in \{0,1\}^\lambda$. This machine is then transformed into a succinct circuit $C'_{s,t}$ (via a transformation due to Pippenger and Fischer [PF79], see Lemma 3.6), and then encoded with our succinct randomized encoding; i.e., $Z = \mathsf{sRE.Enc}(1^\lambda, C'_{s,t}, 0^\lambda, G_{s,t})$ where $C'_{s,t}$ succinctly describes a larger circuit $C_{s,t}$ with $G_{s,t}$ gates and $C_{s,t}$ is equivalent to $M_{s,t}$ (on inputs of size λ). The puzzle solution algorithm simply runs the decoding procedure of the randomized encoding scheme; i.e., $\mathsf{Puz.Sol}(Z)$ outputs $s \leftarrow \mathsf{sRE.Dec}(Z)$.

Security is obtained via reduction to a suitable memory-hard language $\mathcal{L}$. If the security of the constructed puzzle is broken by an adversary $\mathcal{A}$ with small aAT complexity, then we construct a new adversary $\mathcal{B}$ with small aAT complexity which breaks the memory-hard language assumption by deciding whether $x \in \mathcal{L}$ with non-negligible advantage. Suppose that $Z_0 \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_0)$, $Z_1 \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_1)$, b is a random bit, and $t := t(\lambda)$. If $\mathcal{A}(s_0, s_1, Z_b, Z_{1-b})$ can violate the MHP security and predict b with non-negligible probability, then we can construct an algorithm $\mathcal{B}$ with similar aAT complexity that decides our memory-hard language. Algorithm $\mathcal{B}$ first constructs a uniformly succinct circuit $C_{a,a'}$ such that on any input x we have $C_{a,a'}(x) = a$ if $x \in \mathcal{L}$; otherwise $C_{a,a'}(x) = a'$ if $x \notin \mathcal{L}$. By our memory-hard language definition, we can ensure that $C_{a,a'}$ is uniformly succinct and has size $G = t \cdot \mathrm{poly}(\lambda, \log(t))$. Let $C'_{a,a'}$ denote the smaller circuit that succinctly describes $C_{a,a'}$. The adversary computes $Z_i = \mathsf{sRE.Enc}(1^\lambda, C'_{s_i, s_{1-i}}, x, G)$ for $i \in \{0,1\}$, samples $b \xleftarrow{\$} \{0,1\}$, and obtains $b' \leftarrow \mathcal{A}(Z_b, Z_{1-b}, s_0, s_1)$. Our adversary $\mathcal{B}$ outputs 1 if $b = b'$ and 0 otherwise.

Observe that if $x \in \mathcal{L}$ then $\mathsf{Puz.Sol}(Z_0) = s_0$ and $\mathsf{Puz.Sol}(Z_1) = s_1$; otherwise if $x \notin \mathcal{L}$ then $\mathsf{Puz.Sol}(Z_0) = s_1$ and $\mathsf{Puz.Sol}(Z_1) = s_0$. By security of sRE, adversary $\mathcal{A}$ cannot distinguish between $Z_i = \mathsf{sRE.Enc}(1^\lambda, C'_{s_i, s_{1-i}}, x, G)$ and a puzzle generated with $\mathsf{Puz.Gen}$. Thus on input (Z_b, Z_{1-b}, s_0, s_1), the adversary $\mathcal{A}$ outputs $b' = b$ with non-negligible advantage. By our above observation, we have that $\mathcal{B}$ now (probabilistically) decides the memory-hard language $\mathcal{L}$ with non-negligible advantage.

To obtain an adversary $\mathcal{B}'$ that deterministically decides $\mathcal{L}$, we use standard amplification techniques, along with the assumption of $\mathcal{B}'$ being a non-uniform algorithm (à la the argument for $\mathsf{BPP} \subset \mathsf{P/poly}$). Whereas amplification—when performed in parallel—does not significantly increase the total computation depth, *any* amplification increases the aAT complexity of an algorithm by a multiplicative factor proportional to the amount of amplification performed. Intuitively, this is because the aAT complexity of an algorithm A is equal to the sum of aAT complexities of any sub-computations performed by A. See Section 4.1 for discussion.

2.3 Memory-Hard Functions from Memory-Hard Puzzles

Using our new notion of memory-hard puzzles, we construct a one-time memory-hard function under standard cryptographic assumptions (see Section 5). To the best of our knowledge, this is the first such construction in the standard model; i.e., without random oracles [AS15] or other idealized primitives

[CT19]. Recall that informally a function f is memory-hard if any PRAM computing f has large aAT complexity. We define the *one-time* security of a memory-hard function f via the following game between an adversary and an honest challenger. First, an adversary selects an input x and sends it to a challenger. Second, the challenger computes $y_0 = f(x)$ and samples $y_1 \in \{0,1\}^\lambda$ and $b \xleftarrow{\$} \{0,1\}$ uniformly at random, and sends y_b. Then the attacker outputs a guess b' for b. We say that the adversary wins if $b' = b$, and say that f is *ε-one time secure* if the probability that $b' = b$ is at most $\varepsilon(\lambda)$ for security parameter λ. Our construction relies on our new notion of memory-hard puzzles, and additionally uses indistinguishability obfuscation ($i\mathcal{O}$) for circuits and a family of puncturable pseudo-random functions (PPRFs) $\{F_i\}_i$ [BW13, KPTZ13, BGI14]. Informally, PPRFs are pseudo-random functions that allow one to "puncture" a key K at values $x_1, \ldots, x_k$, where the key K can be used to evaluate the function at any point $x \notin \{x_1, \ldots, x_k\}$ and hide the values of the function at the points $x_1, \ldots, x_k$. For any instantiation of $i\mathcal{O}$ that we are aware of our construction is also a (computationally) *data-independent* MHF, i.e., the memory access pattern is (computationally) independent of the secret input x. This is a desirable and useful property that provides natural resistance to side-channel attacks.

We construct our memory-hard function as follows. During the setup phase we generate three PPRF keys K_1, K_2, and K_3 and obfuscate a program $\mathsf{prog}(\cdot,\cdot)$ which on input $(x, \perp)$ outputs a puzzle $\mathsf{Puz.Gen}(1^\lambda, t(\lambda), s; r)$ with solution $s = F_{K_1}(x)$ using randomness $r = F_{K_2}(x)$. On input (x, s') the program checks to see if $s' = F_{K_1}(x)$ and, if so, outputs $F_{K_3}(x)$; otherwise $\perp$. Given the public parameters $\mathsf{pp} = i\mathcal{O}(\mathsf{prog})$ we can evaluate the MHF as follows: (1) run $\mathsf{prog}(x, \perp)$ to obtain a puzzle Z; (2) solve the puzzle Z to obtain $s = \mathsf{Puz.Sol}(Z)$; and (3) run $\mathsf{prog}(x, s)$ to obtain the output $F_{K_3}(x)$. Intuitively, the construction is shown to be one-time memory-hard by appealing to the memory-hard puzzle security, PPRF security, and $i\mathcal{O}$ security.

We establish one-time memory-hardness by showing how to transform an MHF attacker $\mathcal{A}$ into a MHP attacker $\mathcal{B}$ with comparable aAT complexity. Our reduction involves a sequence of hybrids H_0, H_1, H_2 and H_3. Hybrid H_0 is simply our above constructed function. In hybrid H_1 we puncture the PPRF keys $K_i\{x_0, x_1\}$ at target points x_0, x_1 and hard code the corresponding puzzles Z_0, Z_1 along with their solutions—$i\mathcal{O}$ security implies that H_1 and H_0 are indistinguishable. In hybrid H_2 we rely on PPRF security to replace Z_0, Z_1 with randomly generated puzzles independent of the PPRF keys K_1, K_2 and hardcode the corresponding solutions s_0, s_1. Finally, in hybrid H_3 we rely on MHP security to break the relationship between s_i and Z_i; i.e., we flip a coin b' and hardcoded puzzles $Z_0' = Z_{b'}$ and $Z_1' = Z_{1-b'}$ while maintaining $s_i = \mathsf{Puz.Sol}(Z_i)$. In the final hybrid we can show that the attacker cannot win the MHF security game with non-negligible advantage.

Showing indistinguishability of H_2 and H_3 is the most interesting case. In fact, an attacker who can solve either puzzle Z_b or Z_{1-b} can potentially distinguish the two hybrids. Instead, we only argue that the hybrids are indistinguishable if the adversary has small area-time complexity. In particular, if an adversary is able to distinguish between these two hybrids, then we construct an adversary with "small" area-time complexity which breaks the memory-hard puzzle. Note that the barrier to multi-time security occurs in this hybrid as well. Allowing the adversary to evaluate the MHF at multiple distinct inputs, say k of them, increases the aAT of the adversary by a multiplicative factor of k. This could potentially result in an adversary with large enough aAT complexity to simply solve the puzzles Z_b, Z_{1-b}, allowing them to trivially distinguish the hybrids. See Section 5 for details.

2.4 Resource-Bounded Locally Decodable Codes from Cryptographic Puzzles

Recall that a resource-bounded LDC is a locally decodable code that is secure against some class $\mathbb{C}$ of adversaries, assumed to have some resource constraint. For example, $\mathbb{C}$ can be a class of adversaries that are represented by low-depth circuits, or have small (amortized) area-time complexity. In more detail, security of resource-bounded LDCs requires that any adversary in the class $\mathbb{C}$ cannot corrupt an encoding $y = \mathsf{Enc}(x)$ to some $\tilde{y}$ such that (1) the distance between y and $\tilde{y}$ is small; and (2) there exists an index i such that the decoder, when given $\tilde{y}$ as its oracle, outputs x_i with probability less than p.

We construct our resource-bounded LDC by modifying the construction of [BKZ20] to use cryptographic puzzles in place of random oracles. In particular, for algorithm class $\mathbb{C}$, if there exists a cryptographic puzzle that is unsolvable by any algorithm in $\mathbb{C}$, then we use this puzzle to construct a LDC secure against $\mathbb{C}$. Our construction, mirroring [BKZ20], relies on another relaxed LDC: a *private LDC* [OPS07]. Private LDCs are LDCs that are additionally parameterized by a key generation algorithm that on input 1^λ for security parameter λ outputs a shared secret key sk to *both* the encoding and decoding algorithm. Crucially, this secret key is hidden from the adversarial channel.

We construct our Hamming LDC as follows. Let $(\mathsf{Gen}, \mathsf{Enc_p}, \mathsf{Dec_p})$ be a private Hamming LDC. The encoder, on input message x, samples random coins $s \in \{0,1\}^\lambda$ then generates cryptographic puzzle Z with solution s. The encoder then samples a secret key $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda; s)$, where Gen uses random coins s, and encodes the message x as $Y_1 = \mathsf{Enc_p}(x; \mathsf{sk})$. The puzzle Z is then encoded as Y_2 via some repetition code. The encoder then outputs $Y = Y_1 \circ Y_2$. This codeword is corrupted to some $\widetilde{Y}$, which can be parsed as $\widetilde{Y} = \widetilde{Y}_1 \circ \widetilde{Y}_2$. The local decoder, on input index i and given oracle access to $\widetilde{Y}$, first recovers puzzle Z by querying $\widetilde{Y}_2$ (e.g., via random sampling with majority vote). The decoder then solves puzzle Z and recovers solution s. Given s, the local decoder is able to generate the same secret key $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda; s)$ and now runs the local decoder $\mathsf{Dec_p}(i; \mathsf{sk})$. All queries made by $\mathsf{Dec_p}(i; \mathsf{sk})$ are answered by querying $\widetilde{Y}_1$, and the decoder outputs $\mathsf{Dec_p}(i; \mathsf{sk})$. The construction is secure against any class $\mathbb{C}$ for which there exist cryptographic puzzles that are secure against this class. For example, time-lock puzzles give a construction that is secure against the class $\mathbb{C}$ of circuits of low-depth, and memory-hard puzzles give a construction that is secure against the class $\mathbb{C}$ of PRAM algorithms with low aAT complexity.

Security is established via a reduction to the cryptographic puzzle. In particular, if there exists an adversary A in the class $\mathbb{C}$ which can violate the security of our construction, then we construct another adversary in the class $\mathbb{C}$ which can break the security of the cryptographic puzzle. In particular, the reduction relies on a two-phase hybrid distinguishing argument [BKZ20]. Let Enc and Dec be the encoder and local decoder constructed above. Define $\mathsf{Enc}_0 := \mathsf{Enc}$ and define Enc_1 identically as Enc_0, except additionally Enc_1 takes as input a secret key sk_1 (whereas Enc_0 samples a secret key sk_0) that is given to the encoder $\mathsf{Enc_p}(\cdot; \mathsf{sk}_1)$ (whereas Enc_0 gives secret key sk_0). Phase one of the argument samples $b \xleftarrow{\$} \{0,1\}$ uniformly at random to encode a message x with sk_b, and obtains corrupted codeword $\widetilde{Y}_b \leftarrow A(x, \mathsf{Enc}_b(x; \mathsf{sk}_b))$. Let $\widetilde{Y}_b = \widetilde{Y}_{0,b} \circ \widetilde{Y}_{1,b}$ where $\widetilde{Y}_{0,b} = \mathsf{Enc_p}(x; \mathsf{sk}_b)$. Phase two of the argument consists of constructing a distinguisher $\mathcal{D}$ which is given message x, secret key sk_b, and $\widetilde{Y}_{0,b}$. The distinguisher then is given access to the decoder $\mathsf{Dec}_{\mathsf{p},b}^{Y'_{\mathsf{p},b}}$, samples index $i \xleftarrow{\$} [|x|]$, and computes $x'_i \leftarrow \mathsf{Dec}_{\mathsf{p},b}^{Y'_{\mathsf{p},b}}(i; \mathsf{sk}_b)$. It is important to note that the algorithm does not know the bit b at this stage, and needs to to output a guess b' for the bit b. Given this distinguisher, we construct an adversary $B \in \mathbb{C}$ such that B on input (Z_b, Z_{1-b}, s_0, s_1) for uniformly random bit b, where Z_i is a puzzle with solution s_i, outputs b with probability proportional to the distinguisher, breaking the cryptographic puzzle assumption. See Section 6.2 for more details.

3 Preliminaries

Let $\lambda \in \mathbb{N}$ be the security parameter. A function $\mu\colon \mathbb{N} \to \mathbb{R}^+$ is said to be *negligible* if for any polynomial p and all sufficiently large n we have $\mu(n) < 1/|p(n)|$. We let $\mathsf{negl}(\cdot)$ denote the class of negligible functions or an unspecified negligible function. Similarly, we let $\mathrm{poly}(\cdot)$ and $\mathrm{polylog}(\cdot)$ denote the class of polynomial or poly-logarithmic functions, respectively, or unspecified polynomial or poly-logarithmic functions, respectively. For a finite set S we let $x \xleftarrow{\$} S$ denote the process of uniformly sampling elements from S. For positive integer n, we let $[n] := \{1, \ldots, n\}$. We let PPT denote probabilistic polynomial time. For a randomized algorithm A, we let $y \leftarrow A(x)$ denote obtaining output y from A on input x. Sometimes, we fix the coins of A with $r \xleftarrow{\$} \{0,1\}^*$, and denote $y \leftarrow A(x;r)$ as obtaining output y from A using coins r.

3.1 PRAM Algorithms and Area-Time Complexity

We primarily work in the *Parallel Random Access Machine* (PRAM) model. An algorithm A is a *PRAM algorithm* if during each time-step of computation, the algorithm has an internal state and can read multiple positions from memory, perform a computation, then write to multiple positions in memory. For our purposes, it is enough to think of this algorithm as follows: during each time-step, the algorithm makes multiple load requests to memory, makes a small-depth computation (possibly using the loaded values), and write back to multiple locations in memory (see, e.g., [ACK+16, ABP17], for formal definitions).

For a PRAM algorithm A with input $x \in \{0,1\}^*$, we define a *configuration* σ_i as the internal state of A and the non-empty contents of memory at time-step i, and let σ_0 denote the initial configuration of an algorithm A. We define the *trace of* A on input x as $\mathrm{Trace}(A,x) = (\sigma_0, \sigma_1, \ldots \sigma_T)$, where $A(x)$ terminates in T steps. If $A(x)$ does not terminate, we define $\mathrm{Trace}(A,x) := \infty$. We restrict our attention to terminating PRAM algorithms (and thus finite traces). Given $\mathrm{Trace}(A,x)$, we define the *amortized area-time complexity of* A on input x as

$$\mathsf{aAT}(A,x) := \sum_{\sigma \in \mathrm{Trace}(A,x)} |\sigma|.$$

A useful property of aAT complexity is that for two PRAM algorithms A_1, A_2 performing independent computations on x_1 and x_2 respectively then $\mathsf{aAT}((A_1,A_2),(x_1,x_2)) = \mathsf{aAT}(A_1,x_1) + \mathsf{aAT}(A_2,x_2)$; i.e., the aAT cost of running both computations at the same time is the sum of the individual aAT costs. For PRAM algorithm A and for $\lambda \in \mathbb{N}$ we define $\mathsf{aAT}(A,\lambda) := \max_{x \in \{0,1\}^\lambda} \mathsf{aAT}(A,x)$. Finally, for a function $y(\cdot)$ and PRAM algorithm A, we say that $\mathsf{aAT}(A) < y$ if for all $\lambda > 0$ we have $\mathsf{aAT}(A,\lambda) < y(\lambda)$.

We are also concerned with sequential *random access machine* algorithms, or RAM algorithms. A RAM algorithm is simply a PRAM algorithm which during any time-step of the computation only loads from a single location of memory, performs a short computation, and write to a single location of memory. A RAM algorithm running in time t and space s completes its computation after t steps and accesses no more than s cells of memory. It is well-known that a time t RAM algorithm can be simulated by a Turing machine in time $O(t^2)$ (cf., [AB09]).

3.2 Circuits

A Boolean circuit is a function $C\colon \{0,1\}^n \to \{0,1\}^m$ comprised of input gates and a series of AND, OR, and NOT gates with some (possibly bounded) fan-in. We restrict our attention to gates with fan-in 2 (note, NOT always has fan-in 1). and, we let $|C|$ denote the number of (non-input) gates of C (i.e., the size of C). We let $\mathrm{depth}(C)$ denote the depth of C (that is, the longest path from an input gate to an output gate). A *randomized circuit* $C(x;r)$ is a circuit with two types of input wires: wires for the input x and wires for (uniformly) random bits r. For a family of (randomized) circuits $\mathcal{C} = \{C_i\}_{i \in \mathbb{N}}$, we say that the family $\mathcal{C}$ is *uniform* if for every i, there exists an efficient PRAM algorithm which on input i constructs C_i in time $\mathrm{poly}(|C_i|)$. Otherwise $\mathcal{C}$ is *non-uniform*. We are particularly interested in families of *succinct circuits*.

Definition 3.1 (Succinct Circuits [BGT14, GS18]). *Let $C\colon \{0,1\}^n \to \{0,1\}^m$ be a circuit with $N-n$ binary gates. The gates of the circuit are numbered as follows. The input gates are given numbers $\{1,\ldots,n\}$. The intermediate gates are numbered $\{n+1, n+2, \ldots, N-m\}$ such that for any gate g*

with inputs from gates i and j, the label for g is bigger than i and j. The output gates are numbered $\{N-m+1, \dots, N\}$. Each gate $g \in \{n+1, \dots, N\}$ is described by a tuple $(i, j, f_g) \in [g-1]^2 \times \mathsf{GType}$ where the outputs of gates i and j serve as inputs to gate g and f_g denotes the functionality computed by gate g. Here, GType *denotes the set of all binary functions $f : \{0,1\}^2 \to \{0,1\}$.*

We say that C is succinctly describable *if there exists a circuit C^{sc} such that on input $g \in \{n+1, N\}$ outputs description (i, j, f_g) and $|C^{\mathsf{sc}}| < |C|$.*

The above definitions naturally extend to randomized circuits. For notational convenience, for any circuit C^{sc} that succinctly describes a larger circuit C, we define $\mathsf{FullCirc}(C^{\mathsf{sc}}) := C$ and $\mathsf{SuccCirc}(C) := C^{\mathsf{sc}}$. The following lemma states that any Turing machine M is describable by a succinct circuit.

Lemma 3.2 ([PF79, GS18])**.** *Any Turing machine M, which for inputs of size n, requires a maximal running time $t(n)$ and space $s(n)$, can be converted in time $O(|M| + \log(t(n)))$ to a circuit C_{TM} that succinctly represents circuit $C\colon \{0,1\}^n \to \{0,1\}$ where C computes the same function as M (for inputs of size n), and is of size $\widetilde{O}(t(n) \cdot s(n))$.*

We expand Definition 3.1 and define *succinctly describable circuit families.* As with Definition 3.1, the following definition naturally extends to families of randomized circuits.

Definition 3.3 (Uniform Succinct Circuit Families)**.** *We say that a circuit family $\{C_{t,\lambda}\}_{t,\lambda}$ is* succinctly describable *if there exists another circuit family $\{C^{\mathsf{sc}}_{t,\lambda}\}_{t,\lambda}$ such that $|C^{\mathsf{sc}}_{t,\lambda}| = \mathrm{polylog}(|C_{t,\lambda}|)$*[5] *and* $\mathsf{FullCirc}(C^{\mathsf{sc}}_{t,\lambda}) = C_{t,\lambda}$ *for every t, λ. Additionally, if there exists a PRAM algorithm A such that $A(t, \lambda)$ outputs $C^{\mathsf{sc}}_{t,\lambda}$ in time $\mathrm{poly}(|C^{\mathsf{sc}}_{t,\lambda}|)$ for every t, λ, then we say that $\{C_{t,\lambda}\}_{t,\lambda}$ is* uniformly succinct.

3.3 Languages and Decidability

We say that a deterministic algorithm A decides a language $\mathcal{L}$ if for every $x \in \mathcal{L}$, we have $A(x) = 1$ (and $A(x) = 0$ for $x \notin \mathcal{L}$). We say that a randomized algorithm A *ε-decides* a language $\mathcal{L}$ if for every $x \in \mathcal{L}$ we have $\Pr[A(x) = 1] \geqslant 1/2 + \varepsilon(|x|)$, and for every $x \notin \mathcal{L}$ we have $\Pr[A(x) = 0] \geqslant 1/2 + \varepsilon(|x|)$. Similarly, we say that $\mathcal{L}_i := \mathcal{L} \cap \{0,1\}^i$ is decided by algorithm A if A restricted to i-bit inputs decides $\mathcal{L}_i$.

We say that a language $\mathcal{L}$ is decided by circuit family $\{C_i\}_{i \in \mathbb{N}}$ if for every i the language $\mathcal{L}_i$ is decided by C_i. We say that a randomized circuit family $\{C_i\}_{i \in \mathbb{N}}$ *ε-decides* a language $\mathcal{L}$ if for every $i \in \mathbb{N}$ and every $x \in \{0,1\}^i$, the circuit C_i ε-decides the language $\mathcal{L}_i$ where the probability is taken over uniformly random string $r \in \{0,1\}^*$.

3.4 Cryptographic Primitives

We assume the existence of puncturable pseudorandom functions (PRFs) and indistinguishability obfuscation. It is well-known that one-way functions imply the existence of (puncturable) PRFs [GGM86, HILL99]. We provide brief definitions here and refer the reader to Appendix A for formal definitions.

Informally, a punctured PRF family can efficiently generate a punctured key $K \setminus \{x_1, \dots, x_k\}$ which can be used to evaluate $F(K, x)$ on any input $x \notin \{x_1, \dots, x_k\}$ and hide the values $F(K, x_1), \dots, F(K, x_k)$.

For a circuit class $\mathcal{C} = \{\mathcal{C}_\lambda\}_\lambda$ we say that a PPT algorithm $i\mathcal{O}$ is an *indistinguishability obfuscator* for $\mathcal{C}$ if (1) for every λ and every $C \in \mathcal{C}_\lambda$, we have that $C'(x) = C(x)$ for every x and $C' \leftarrow i\mathcal{O}(1^\lambda, C)$; and (2) if $C_0, C_1 \in \mathcal{C}_\lambda$ compute the same functionality, then no PPT adversary can distinguish between $i\mathcal{O}(1^\lambda, C_0)$ and $i\mathcal{O}(1^\lambda, C_1)$ except with negligible advantage.

[5] For our purposes, we require the size of the succinct circuit to be poly-logarithmic in the size of the full circuit. One can easily replace this requirement with the requirement presented in Definition 3.1.

Finally, a *randomized encoding* scheme [IK00] is described by a pair of algorithms $\mathsf{RE} = (\mathsf{RE.Enc}, \mathsf{RE.Dec})$, where $\mathsf{RE.Enc}$ is a randomized algorithm that on input 1^λ, the description of a machine M, input x, and time bound t outputs an encoding $\widehat{M}_x$ and $\mathsf{RE.Dec}$ is a deterministic algorithm that on input $\widehat{M}_x$ outputs y, where y is the output of $M(x)$ after t steps of computation. In this work, we extensively make use of *succinct randomized encodings*. For our purposes, we require succinct randomized encodings for succinct circuits and define these encodings directly.

Definition 3.4 ([BGL+15, GS18]). *A* succinct randomized encoding *consists of two algorithms* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *with the following syntax:*

- $\mathsf{sRE.Enc}(1^\lambda, C', x, G)$*: takes as input the security parameter* λ*, a succinct circuit* C' *encoding a larger circuit* C*, input* x *and size* G *(gates) of the circuit* C*, and outputs the randomized encoding* $\widehat{C}_{x,G}$*.*
- $\mathsf{sRE.Dec}(\widehat{C}_{x,G})$*: takes as input the randomized encoding* $\widehat{C}_{x,G}$ *and deterministically computes the output* y*.*

We require the scheme to satisfy the following three properties.

Correctness. *For every* x *and* C' *such that* $\mathsf{FullCirc}(C') = C$ *and* $|C| = G$*, it holds that* $y = C(x)$ *with probability* 1 *over the random coins of* $\mathsf{sRE.Enc}$*.*

Security. *There exists a PPT simulator* Sim *such that for any poly-size adversary* $\mathcal{A}$ *there exists negligible function* μ *such that for every* $\lambda \in \mathbb{N}$*, circuit* C' *encoding larger circuit* C *with* G *gates, and input* x*:*

$$\left|\Pr[\mathcal{A}(\widehat{C}_{x,G}) = 1] - \Pr[\mathcal{A}(\mathsf{Sim}(1^\lambda, y, C', G, 1^{|x|})) = 1]\right| \leqslant \mu(\lambda),$$

where $\widehat{C}_{x,G} \leftarrow \mathsf{sRE.Enc}(1^\lambda, C', x, G)$ *and* y *is the output of* $C(x)$*.*

Succinctness. *The running time of* $\mathsf{sRE.Enc}$ *on a sequential RAM and the size of the encoding* $\widehat{C}_{x,G}$ *are* $\mathrm{poly}(|C'|, |x|, \log(G), \lambda)$*. The running time of* $\mathsf{sRE.Dec}$ *on a sequential RAM is* $G \cdot \mathrm{poly}(\log(G), \lambda)$*.*[6]

We are also interested in *concretely secure* succinct randomized encodings, as opposed to the *asymptotic security* definition given in Definition 3.4.

Definition 3.5 ((t, s, ε)-Secure Succinct Randomized Encoding). *A succinct randomized encoding* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *is* $(t(\cdot), s(\cdot), \varepsilon(\cdot))$-secure *if it satisfies the following concrete security requirement: there exists a probabilistic simulator* Sim *and a polynomial* $p(\cdot)$ *such that for every security parameter* λ*, every adversary* $\mathcal{A}$ *running in time at most* $t(\lambda)$ *and every circuit* C' *representing a larger circuit* C *with* $G \leqslant s(\lambda)$ *gates and every input* $x \in \{0,1\}^\lambda$*:*

$$\left|\Pr[\mathcal{A}(\widehat{C}_{x,G}) = 1] - \Pr[\mathcal{A}(\mathsf{Sim}(1^\lambda, y, C', G)) = 1]\right| \leqslant \varepsilon(\lambda),$$

where $\widehat{C}_{x,G} \leftarrow \mathsf{sRE.Enc}(1^\lambda, C', x, G)$*,* y *is the output of* $C(x)$*, and* Sim *runs in time at most* $G \cdot p(\lambda)$*.*

Note that one-way functions and $i\mathcal{O}$ are sufficient for constructing succinct randomized encodings [KLW15, BGL+15]. In our constructions, we use the succinct randomized encoding scheme of Garg and Srinivasan [GS18], with a slight modification. The encoding scheme is modified to accept uniformly succinct circuits as inputs rather than Turing machines. The encoding scheme of [GS18] transforms the input Turing machine to obtain a succinct circuit via Lemma 3.2, then runs $i\mathcal{O}$ on this succinct circuit. We observe that we can directly input the succinct circuit to avoid the need of this internal transformation.

[6] A weaker requirement succinctness requirement allows for the running time of $\mathsf{sRE.Dec}$ to be $\mathrm{poly}(G, \lambda)$. This stronger requirement is achievable and is crucial for applications to cryptographic puzzles.

Lemma 3.6 ([GS18]). *Assuming the existence of $i\mathcal{O}$ for circuits and somewhere statistically binding hash functions, there exists a succinct randomized encoding* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *for succinct circuits C' with G' gates representing larger circuit C with G gates such that* $\mathsf{sRE.Enc}$ *runs in time* $\mathrm{poly}(G', \log(G), \lambda, n)$ *and* $\mathsf{sRE.Dec}$ *runs in time* $G \cdot \mathrm{poly}(G', \log(G), \lambda)$*, where n is the input length of C.*

4 Memory-Hard Puzzles

We formally introduce and define the notion of *memory-hard puzzles.* First we define puzzles.

Definition 4.1 (Puzzles [BGJ+16]). *Let $\lambda \in \mathbb{N}$ be the security parameter. A pair of algorithms* $(\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *is a* puzzle *if they satisfy the following requirements.*

- $\mathsf{Puz.Gen}(1^\lambda, t, s)$ *is a randomized algorithm which takes as input security parameter $\lambda \in \mathbb{N}$, time parameter $t := t(\lambda) < 2^\lambda$, and arbitrary solution $s \in \{0,1\}^\lambda$ and outputs a puzzle Z.*
- $\mathsf{Puz.Sol}(Z)$ *is a deterministic algorithm which takes puzzle Z as input and outputs solution s.*
- ***Completeness**: For every $\lambda \in \mathbb{N}$, $t < 2^\lambda$, $s \in \{0,1\}^\lambda$, and puzzle $Z \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s)$, we have that $s = \mathsf{Puz.Sol}(Z)$ with probability 1 over the random coins of* $\mathsf{Puz.Gen}$*.*
- ***Efficiency**: For all λ, t, s, we require that $\mathsf{Puz.Gen}(1^\lambda, t, s)$ is computable in time $\mathrm{poly}(\lambda, \log(t))$ on a sequential RAM, and $\mathsf{Puz.Sol}(Z)$ is computable in time $t \cdot \mathrm{poly}(\lambda)$ on a sequential RAM.*

We remark that in the above definition we are interested in the case that $t(\lambda)$ is a polynomial, and without loss of generality we assume that $\mathsf{Puz.Gen}$ uses λ-bits of randomness.

Informally, we define a memory-hard puzzle as a puzzle that requires any PRAM algorithm solving it to have high aAT complexity. Formally, we introduce two flavors of memory-hard puzzles. First we consider a memory-hard puzzle with an asymptotic security definition.

Definition 4.2 (g-Memory Hard Puzzle). *A puzzle* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *is a g-*memory hard puzzle *if there exists a polynomial t' such that for all polynomials $t > t'$ and for every PRAM algorithm $\mathcal{A}$ with $\mathsf{aAT}(\mathcal{A}) < y$ for $y(\lambda) := g(t(\lambda), \lambda)$, there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$ and every pair $s_0, s_1 \in \{0,1\}^\lambda$ we have*

$$\left| \Pr\left[\mathcal{A}(Z_b, Z_{1-b}, s_0, s_1) = b\right] - \frac{1}{2} \right| \leqslant \mu(\lambda), \tag{1}$$

where the probability is taken over $b \xleftarrow{\$} \{0,1\}$ and $Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ for $i \in \{0,1\}$.

Note that for any difficulty parameter $t := t(\lambda)$ for security λ, we assume that $\mathsf{Puz.Sol}$ is computable in time $t \cdot \mathrm{poly}(\lambda)$ on a sequential RAM algorithm. This implies that there exists a PRAM algorithm A computing $\mathsf{Puz.Sol}$ has $\mathsf{aAT}(A, \lambda) \leqslant (t \cdot \mathrm{poly}(\lambda))^2 = t^2 \cdot \mathrm{poly}(\lambda)$. This yields an upper bound on the function g of Definition 4.2: take t to be any (large enough) polynomial. Then suitable values of g (ignoring $\mathrm{poly}(\lambda)$ factors) include $g = t^2/\log(t)$ or $g = t^{2-\theta}$ for small constant $\theta > 0$. In particular, $g = o(t^2)$ is necessary for any memory-hard puzzle (by our definitions).

We complement Definition 4.2 with the following concrete security definition.

Definition 4.3 ((g, ε)-Memory Hard Puzzle). *A puzzle* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *is a (g, ε)-*memory hard puzzle *if there exists a polynomial t' such that for all polynomials $t > t'$ and every PRAM algorithm $\mathcal{A}$ with $\mathsf{aAT}(\mathcal{A}) < y$ for $y(\lambda) := g(t(\lambda), \lambda)$, and for all $\lambda > 0$ and any pair $s_0, s_1 \in \{0,1\}^\lambda$, we have*

$$\left| \Pr\left[\mathcal{A}(Z_b, Z_{1-b}, s_0, s_1) = b\right] - \frac{1}{2} \right| \leqslant \varepsilon(\lambda), \tag{2}$$

where the probability is taken over $b \xleftarrow{\$} \{0,1\}$ *and* $Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ *for* $i \in \{0,1\}$*. If* $\varepsilon(\lambda) = 1/\operatorname{poly}(\lambda)$*, we say the puzzle is* weakly memory-hard*.*

Similar to Definition 4.2, suitable values of g for Definition 4.3 include $g = t^2/\log(t)$ and $g = t^{2-\theta}$ for small constant $\theta > 0$, as any PRAM algorithm with aAT complexity at least t^2 can compute $\mathsf{Puz.Sol}$ for difficulty parameter t.

We construct memory-hard puzzles from standard cryptographic assumptions and the (essentially minimal) assumption that a *memory-hard language* exists. This assumption is similar to the non-parallelizing language assumption necessary for constructing the time-lock puzzles of Bitansky et al. [BGJ+16]. Our first step towards defining memory-hard languages is defining a language class that is decidable by uniformly succinct circuit families.

Definition 4.4 (Language Class SC_t)**.** *Let t be a positive function. We define SC_t as the class of languages $\mathcal{L}$ decidable by a uniformly succinct circuit family $\{C_{t,\lambda}\}_\lambda$ (as per Definition 3.3) such that there exists a polynomial p satisfying $|C_{t,\lambda}| \leqslant t \cdot p(\lambda, \log(t))$ for every λ and $t := t(\lambda)$.*

Given the language class SC_t, we now define memory-hard languages.

Definition 4.5 ((g,ε)-Memory Hard Language)**.** *Let t be a positive function. A language $\mathcal{L} \in \mathsf{SC}_t$ is a (g,ε)-memory hard language if for every PRAM algorithm $\mathcal{B}$ with $\mathsf{aAT}(\mathcal{B},\lambda) < g(t(\lambda),\lambda)$, the algorithm $\mathcal{B}$ does not $\varepsilon(\lambda)$-decide $\mathcal{L}_\lambda$ for every λ. If $\varepsilon(\lambda) \in (0,1/2)$ is a constant, we say $\mathcal{L}$ is* weakly memory-hard*. If $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$, we say $\mathcal{L}$ is* strongly memory-hard*.*

Remark 4.6. One can also define weakly memory-hard languages for $\varepsilon(\lambda) = 1/\operatorname{poly}(\lambda)$; however, this is essentially equivalent to our above definition of weakly memory-hard languages. Given a (g,ε)-memory hard language $\mathcal{L}$ for $\varepsilon(\lambda) = 1/\operatorname{poly}(\lambda)$, the language $\mathcal{L}$ is also a (g',ε')-memory hard language for $g'(t(\lambda),\lambda) = g(t(\lambda),\lambda) \cdot \Theta(1/\varepsilon(\lambda))$ and any constant $\varepsilon' \in (0,1/2)$. This can be seen as follows: given any adversary $\mathcal{B}$ with $\mathsf{aAT}(\mathcal{B},\lambda) < g(t(\lambda),\lambda)$ that ε-decides $\mathcal{L}_\lambda$, we can construct an adversary $\mathcal{B}'$ with $\mathsf{aAT}(\mathcal{B}',\lambda) < g'(t(\lambda),\lambda)$ that ε'-decides $\mathcal{L}_\lambda$. The adversary $\mathcal{B}'$ simply runs $\mathcal{B}$ in parallel $\Theta(1/\varepsilon(\lambda))$ times and takes the majority output, where the hidden constant depends on the target constant $\varepsilon' > 0$.

Note that while parallel amplification does not increase the overall depth of a computation both sequential and parallel amplification incur significant overheads to the aAT complexity of the algorithm performing the amplification. This is simply due to a key property of aAT complexity: if a PRAM algorithm A is running two computations A_1 and A_2 with inputs x_1, x_2, respectively, then $\mathsf{aAT}(A,(x_1,x_2)) = \mathsf{aAT}(A_1,x_1) + \mathsf{aAT}(A_2,x_2)$. In fact, this is exactly what occurs when a PRAM algorithm performs amplification: it is repeating a computation some number, say $c \in \mathbb{N}$, times, then takes the majority of all the outputs of all these computations. This incurs a multiplicative blow-up by c in the aAT complexity of the PRAM algorithm (plus the aAT complexity of computing majority).

Defining the correct machine model for memory-hard languages is surprisingly subtle. While we require our memory-hard languages to be decidable by uniformly succinct circuits, one can imagine a simpler definition where we require decidability with respect to single-tape Turing machines (TMs) à la [BGJ+16]. In the context of time-lock puzzles, there are plausible sequentially hard languages that can be decided in time $t := t(\lambda)$ on a single-tape TM; e.g., the language $\mathcal{L}_{\mathsf{Puz}} := \{(N,x,t) \colon \exists y \text{ s.t. } y = x^{2^t} \bmod N\}$, where N is the product of two safe primes [RSW96] can be decided in time $t \cdot \operatorname{polylog}(N)$ on a single-tape TM. However, in Section 8 we show that *any* language $\mathcal{L}$ that can be decided by a single-tape TM in time t can also be decided by any PRAM algorithm with aAT complexity at least $\Omega(t^{1.8} \cdot \log(t))$. Thus if SC_t is defined with respect to single-tape Turing machines, this result rules out any (g,ε)-memory hard language for $g = \Omega(t^{1.8} \cdot \log(t))$ and *any* ε.

On the positive side, we show that SC_t with respect to uniformly succinct circuits is essentially *minimal*, and also *plausible*. Section 4.2 discusses the minimality of Definitions 4.4 and 4.5 and Section 7 discusses the plausibility of the existence of such memory-hard languages. We stress that barring any major advances in circuit complexity lower bounds, it is highly unlikely that we will be able to formally prove the existence of memory-hard languages. For now, we turn to presenting our constructions of memory-hard puzzles.

4.1 Memory-Hard Puzzle Construction

We present our construction of memory-hard puzzles. Our construction relies on the succinct randomized encoding scheme sRE for succinct circuits given by Lemma 3.6.

Construction 4.7. *Let* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *be a succinct randomized encoding for succinct circuits, let* $\lambda \in \mathbb{N}$ *be the security parameter, let* t *be polynomial in* λ*, and let* $s \in \{0,1\}^{\lambda}$.

- $\mathsf{Puz.Gen}(1^{\lambda}, t, s)$*: On input* $1^{\lambda}, t$*, and* s*, the algorithm first defines a Turing machine* $M_{t,s}$ *which on any input* x *outputs* s *after delaying for* t *steps. The algorithm then applies Lemma 3.2 to construct a circuit* $C^{\mathsf{sc}}_{t,s,\lambda}$ *which succinctly represents a larger circuit* $C_{t,s,\lambda}$ *equivalent to* $M_{t,s}$ *on inputs of size* λ*. Finally the algorithm outputs* $Z \leftarrow \mathsf{sRE.Enc}(1^{\lambda}, C^{\mathsf{sc}}_{t,s,\lambda}, 0^{\lambda}, t)$.
- $\mathsf{Puz.Sol}(Z)$*: On input* Z*, the algorithm outputs output* $s \leftarrow \mathsf{sRE.Dec}(Z)$.

We prove that Construction 4.7 satisfies both of our notions of memory-hard puzzles, depending on the flavor of the security of the succinct randomized encoding scheme sRE. In either case, we use the succinct randomized encoding scheme of Lemma 3.6. First, assuming a asymptotically secure succinct randomized encoding scheme sRE (Definition 3.4) and the existence of a *strong* memory-hard language, we obtain an asymptotically secure memory-hard puzzle.

Theorem 4.8. *Let* t *be a polynomial and let* g *be a function. Let* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *be a succinct randomized encoding scheme. If there exists a* g'*-strong memory-hard language* $\mathcal{L} \in \mathsf{SC}_t$ *for*

$$g'(t(\lambda), \lambda) := g(t(\lambda), \lambda) + 2 \cdot p_{\mathsf{sRE}}\big(\log(t(\lambda)), \lambda\big)^2 + 2 \cdot p_{\mathsf{SC}}\big(\log(t(\lambda)), \log(\lambda)\big)^2 + O(\lambda),$$

then Construction 4.7 is a g*-memory hard puzzle. Here,* p_{sRE} *and* p_{SC} *are fixed polynomials for the run-times of* $\mathsf{sRE.Enc}$ *and the uniform machine constructing the uniform succinct circuit of* $\mathcal{L}$*, respectively.*

To get a handle on Theorem 4.8, consider a large enough polynomial t such that $t \gg p_{\mathsf{sRE}}(\log(t), \lambda)$ and $t \gg p_{\mathsf{SC}}(\log(t), \log(\lambda))$. Then if there exists a g'-strong MHL for $g'(t, \lambda) = t^2/\log(t)$, we obtain a g-memory hard puzzle for $g(t, \lambda) = (1 - o(1)) \cdot g'(t, \lambda)$ (i.e., there is little loss in the memory-hardness of the constructed puzzle).

Next, assuming a concretely secure succinct randomized encoding scheme sRE (Definition 3.5) and the existence of a *weak* memory-hard language, we obtain a weakly secure memory-hard puzzle.

Theorem 4.9. *Let* t *be a polynomial and let* g *be a function. Let* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *be a* $(g, s, \varepsilon_{\mathsf{sRE}})$*-secure succinct randomized encoding scheme for* $g := g(t(\lambda), \lambda)$ *and* $s(\lambda) := t(\lambda) \cdot \mathrm{poly}(\lambda, \log(t(\lambda)))$ *such that* p_{sRE} *is a fixed polynomial for the runtime of* $\mathsf{sRE.Enc}$*. Let* $\varepsilon(\lambda) = 1/\mathrm{poly}(\lambda) > 3\varepsilon_{\mathsf{sRE}}(\lambda)$ *be fixed.*

If there exists a $(g', \varepsilon_{\mathcal{L}})$*-weakly memory-hard language* $\mathcal{L} \in \mathsf{SC}_t$ *for*

$$g'(t(\lambda), \lambda) := \big[g(t(\lambda), \lambda) + 2 \cdot p_{\mathsf{sRE}}\big(\log(t(\lambda)), \lambda\big)^2 + 2 \cdot p_{\mathsf{SC}}\big(\log(t(\lambda)), \log(\lambda)\big)^2 + O(\lambda)\big] \cdot \Theta(1/\varepsilon(\lambda)),$$

and some constant $\varepsilon_{\mathcal{L}} \in (0, 1/2)$*, then Construction 4.7 is a* (g, ε)*-weakly memory-hard puzzle. Here,* p_{SC} *is a fixed polynomial for the runtime of the uniform machine constructing the uniform succinct circuit for* $\mathcal{L}$*.*

Notice here that we lose a factor of $\Theta(1/\varepsilon)$ when compared with Theorem 4.8. Concretely, using our same example from Theorem 4.8, if t is sufficiently large such that $t \gg p_{\mathsf{sRE}}(\log(t), \lambda)$ and $t \gg p_{\mathsf{SC}}(\log(t), \log(\lambda))$, and if $\varepsilon = 1/\lambda^2$, then for $g' = t^2/\log(t)$ we obtain a (g, ε)-weakly memory-hard puzzle for $g = g' \cdot \Theta(\lambda^2)$. This loss is due to the security reduction: our adversary performs amplification to boost the success probability of breaking the weakly memory-hard language assumption from ε to the constant $\varepsilon_{\mathcal{L}}$. To hit the constant $\varepsilon_{\mathcal{L}}$, one needs to amplify $\Theta(1/\varepsilon)$ times. As discussed previously, amplification directly incurs a multiplicative blow-up in the aAT complexity of a PRAM algorithm performing the amplification.

Remark 4.10 (Non-uniform PRAM Algorithms). If we modify our memory-hard language definition (Definition 4.5) to allow for *non-uniform* PRAM algorithm adversaries, then Theorem 4.9 holds with respect to $\varepsilon_{\mathcal{L}} = 1/2$. With a non-uniform algorithm, we perform amplification à la $\mathsf{BPP} \subset \mathsf{P}/\operatorname{poly}$ in our security reduction to obtain an adversary which breaks the memory-hard language assumption. This results in an additional $\operatorname{poly}(\lambda)$ blow-up in the aAT complexity upper bound g. This is in contrast to Bitansky et al. [BGJ+16]: as they are concerned with the depth of the computation (as opposed to the aAT complexity), parallel amplification does not increase the overall depth of the computation.

Efficiency and Correctness of Construction 4.7. Let $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ be the puzzle of Construction 4.7 and let $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ be the succinct randomized encoding used in the construction. Correctness directly follows by correctness of the succinct randomized encoding scheme. For efficiency, first consider the generation algorithm $\mathsf{Puz.Gen}$. On input $1^\lambda, t, s$, the Turing machine generated by $\mathsf{Puz.Gen}$ has description size $O(\lambda + \log(t))$, runs in time t and space $O(\lambda + \log(t))$, and can be generated in time $O(\lambda + \log(t))$. By Lemma 3.2, the circuit $C_{t,s,\lambda}$ equivalent to $M_{t,s}$ has size $t \cdot \operatorname{poly}(\lambda, \log(t))$, and thus the succinct circuit $C^{\mathsf{sc}}_{t,s,\lambda}$ representing $C_{t,s,\lambda}$ has size $\operatorname{poly}(\lambda, \log(t))$. Next $\mathsf{Puz.Gen}$ obtains $Z \leftarrow \mathsf{sRE.Enc}(1^\lambda, C^{\mathsf{sc}}_{t,s,\lambda}, 0^\lambda, t)$. By definition, $\mathsf{sRE.Enc}(1^\lambda, C, x, G)$ runs in sequential time $\operatorname{poly}(|C|, |x|, \log(G), \lambda)$ for any succinct circuit C such that $|\mathsf{FullCirc}(C)| = G$, input x, and security parameter λ. This implies that $\mathsf{sRE.Enc}(1^\lambda, C^{\mathsf{sc}}_{t,s,\lambda}, 0^\lambda, t)$ runs in time $\operatorname{poly}(\lambda, \log(t))$. Thus the overall efficiency of $\mathsf{Puz.Gen}$ is $\operatorname{poly}(\lambda, \log(t)) + O(\lambda + \log(t)) = \operatorname{poly}(\lambda, \log(t))$ as desired.

Now consider the solve algorithm $\mathsf{Puz.Sol}$. On input $Z \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s)$, the algorithm $\mathsf{Puz.Sol}$ simply computes and outputs $s \leftarrow \mathsf{sRE.Dec}(Z)$. By definition of $\mathsf{Puz.Gen}$, we have that $Z \leftarrow \mathsf{sRE.Enc}(1^\lambda, C^{\mathsf{sc}}_{t,s,\lambda}, 0^\lambda, t)$, where $C^{\mathsf{sc}}_{t,s,\lambda}$ is the succinct circuit described above. By Lemma 3.6 the algorithm $\mathsf{sRE.Dec}(Z)$ runs in time $G \cdot \operatorname{poly}(G', \log(G), \lambda)$, where $|C_{t,s,\lambda}| = G$ and $|\mathsf{SuccCirc}(C_{t,s,\lambda})| = G'$. Further, we have $|C_{t,s,\lambda}| = t \cdot \operatorname{poly}(\lambda, \log(t))$, and by assumption (Definition 4.4) we have $G' = \operatorname{polylog}(G) = \operatorname{polylog}(\lambda, t)$. This implies that $\mathsf{sRE.Dec}$ runs in time $t \cdot \operatorname{poly}(\lambda, \log(t))$. Finally, recalling that t is a polynomial in λ, we have that $\mathsf{sRE.Dec}$ runs in time $t \cdot \operatorname{poly}(\lambda)$, which implies that $\mathsf{Puz.Sol}$ runs in time $t \cdot \operatorname{poly}(\lambda)$ as desired.

Memory-Hardness of Construction 4.7. We give a high-level overview of the proof of memory-hardness of Construction 4.7, and present the full proofs in Appendices B and C. Focusing first on Theorem 4.8, we argue memory-hardness via a reduction: if there exists an adversary A with aAT complexity less than g such that A's advantage in Eq. (1) is at least $1/\operatorname{poly}(\lambda)$ (thus violating Definition 4.2), then we can construct an adversary B with aAT complexity less than g' which is able to decide the language $\mathcal{L} \in \mathsf{SC}_t$ with non-negligible advantage (i.e., $1/\operatorname{poly}(\lambda)$ advantage).

The reduction proceeds as follows: suppose we have a PRAM adversary A with aAT complexity less than g such that A has at least $1/\operatorname{poly}(\lambda)$ advantage in Eq. (1). By assumption,since $\mathcal{L} \in \mathsf{SC}_t$, there exists PRAM algorithm $A_{\mathcal{L}}$ which on input t, λ outputs a succinct circuit $C^{\mathsf{sc}}_{\mathcal{L}}$ such that the circuit $C_{\mathcal{L}} = \mathsf{FullCirc}(C^{\mathsf{sc}}_{\mathcal{L}})$ decides the language $\mathcal{L}$. Given $C_{\mathcal{L}}$, we define a circuit $\widetilde{C}_{a,b}\colon \{0,1\}^\lambda \to \{0,1\}^\lambda$ for any $a, b \in \{0,1\}^\lambda$ such that

$$\widetilde{C}_{a,b}(x) = \begin{cases} a & C_{\mathcal{L}}(x) = 1 \\ b & C_{\mathcal{L}}(x) = 0 \end{cases}.$$

The key observation is that for fixed $a, b \in \{0,1\}^\lambda$, the circuit $\widetilde{C}_{a,b}$ is a uniformly succinct circuit; that is, there exists a PRAM algorithm $\widetilde{A}$ which on input t, λ, a, b outputs a succinct circuit $\widetilde{C}^{\mathsf{sc}}_{a,b}$ such that $\widetilde{C}_{a,b} = \mathsf{FullCirc}(\widetilde{C}^{\mathsf{sc}}_{a,b})$. Note that the circuit $\widetilde{C}_{a,b}$ can be simply described as the circuit $C_{\mathcal{L}}$, 2λ fixed input gates for the values $a, b \in \{0,1\}^\lambda$, and 2λ AND-gates which output a or b depending on if $C_{\mathcal{L}}(x)$ is 1 or 0, respectively. So $\widetilde{C}_{a,b}$ is a circuit of size $|C_{\mathcal{L}}| + O(\lambda) \leqslant t \cdot \mathrm{poly}(\lambda, \log(t)) + O(\lambda) = t \cdot \mathrm{poly}(\lambda, \log(t))$ (which is asymptotically the same as $C_{\mathcal{L}}$). Thus the PRAM algorithm $\widetilde{A}$ simply runs $A_{\mathcal{L}}$ to obtain $C^{\mathsf{sc}}_{\mathcal{L}}$ and adjusts this circuit to include these fixed wire values and the AND-gates.

Crucially, we use the PRAM algorithm $\widetilde{A}$ to construct an input to the adversary A such that if A can distinguish this input then we can use the output of A to decide the language $\mathcal{L}$. Looking at the definition of $\mathsf{Puz.Gen}$ of Construction 4.7, we see that $\mathsf{Puz.Gen}$ outputs a succinct randomized encoding Z of a succinct circuit $C^{\mathsf{sc}}_{t,s,\lambda}$ with full circuit $\mathsf{FullCirc}(C^{\mathsf{sc}}_{t,s,\lambda})$ of size size $\widetilde{O}(t)$ (note that polylog factors are hidden). Recall that the succinct circuit $\widetilde{C}^{\mathsf{sc}}_{a,b}$ that is output by $\widetilde{A}$ represents a circuit $\widetilde{C}_{a,b} = \mathsf{FullCirc}(\widetilde{C}^{\mathsf{sc}}_{a,b})$ with size at most $t \cdot \mathrm{poly}(\lambda, \log(t)) = \widetilde{O}(t)$ since t is a polynomial in λ. This implies that $|\mathsf{sRE.Enc}(\widetilde{C}^{\mathsf{sc}}_{a,b})| \approx |\mathsf{sRE.Enc}(C^{\mathsf{sc}}_{t,s,\lambda})|$.

Putting it all together, we construct randomized PRAM algorithm B which decides $\mathcal{L}$ with non-negligible advantage as follows. Let $s_0, s_1 \in \{0,1\}^\lambda$ be the puzzle solutions given as part of the input to A.

1. Algorithm B first constructs two succinct circuits $\widetilde{C}^{\mathsf{sc}}_i := \widetilde{C}_{s_i, s_{1-i}}$ for $i \in \{0,1\}$ using the PRAM algorithm $\widetilde{A}$.
2. Next, algorithm B computes $Z_i \leftarrow \mathsf{sRE.Enc}(1^\lambda, \widetilde{C}^{\mathsf{sc}}_i, 0^\lambda, t)$ for $i \in \{0,1\}$.
3. Algorithm B then samples $b \xleftarrow{\$} \{0,1\}$ and obtains $b' \leftarrow A(Z_b, Z_{1-b}, s_0, s_1)$.
4. Algorithm B outputs 1 if and only if $b = b'$; otherwise it outputs 0.

Note that if $x \in \mathcal{L}$ then $\mathsf{sRE.Dec}(Z_i) = \mathsf{Puz.Sol}(\mathsf{Puz.Gen}(s_i))$, and if $x \notin \mathcal{L}$ then $\mathsf{sRE.Dec}(Z_i) = \mathsf{Puz.Sol}(\mathsf{Puz.Gen}(s_{1-i}))$ for $i \in \{0,1\}$. Thus it holds that

$$\begin{aligned} (Z_b, Z_{1-b}, s_0, s_1) &\equiv (\mathsf{Puz.Gen}(s_b), \mathsf{Puz.Gen}(s_{1-b}), s_0, s_1) && \text{if } x \in \mathcal{L} \\ (Z_b, Z_{1-b}, s_0, s_1) &\equiv (\mathsf{Puz.Gen}(s_{1-b}), \mathsf{Puz.Gen}(s_b), s_0, s_1) && \text{if } x \notin \mathcal{L}. \end{aligned}$$

Now since $|\mathsf{sRE.Enc}(\widetilde{C}^{\mathsf{sc}}_{a,b})| \approx |\mathsf{sRE.Enc}(C^{\mathsf{sc}}_{t,s,\lambda})|$, we can appeal to the security of the succinct randomized encoding and the assumption that A distinguishes with non-negligible advantage at least $\varepsilon(\lambda) = 1/\mathrm{poly}(\lambda)$. That is, we have that the adversary B decides the language $\mathcal{L}$ with probability at least $1 - (\varepsilon(\lambda) - \mu(\lambda))$, where μ is a fixed negligible function given by the security of the randomized encoding. At this point, we are almost done: the algorithm B decides the language $\mathcal{L}$ with non-negligible advantage. The final step is arguing that $\mathsf{aAT}(B, \lambda) < g'(t(\lambda), \lambda)$. This can be seen by observing that the aAT complexity of B is proportional to the aAT complexity of A, $\mathsf{Puz.Gen}$, and the PRAM algorithm for the succinct circuits of SC_t. Thus we obtain a PRAM adversary B which violates the g'-strong memory-hard language assumption on $\mathcal{L}$.

For Theorem 4.9, the proof is nearly identical except we appeal to the concrete security of the succinct randomized encoding. By carefully specifying the parameters of the randomized encoding scheme, we obtain the same adversary B which decides the language $\mathcal{L}$ with advantage $1/\operatorname{poly}(\lambda)$. The final step is then constructing adversary $\mathcal{B}$ which decides $\mathcal{L}$ with constant advantage in the range $(0, 1/2)$. This is done via amplification à la $\mathsf{BPP} \subset \mathsf{P}/\operatorname{poly}$; namely, $\mathcal{B}$ runs B in parallel $\Theta(1/\varepsilon)$ times and takes the majority output, which results in an adversary with constant advantage $< 1/2$. Further, the adversary $\mathcal{B}$ with aAT complexity that is a factor $\Theta(1/\varepsilon)$ larger than the aAT complexity of B, completing the proof.[7]

Remark 4.11. As noted many times, amplification directly incurs a multiplicative blow-up in the aAT complexity of a PRAM algorithm performing the amplification. Thus we must carefully control the number of times our adversary performs amplification in the security reduction, else the newly constructed adversary would have aAT complexity too large, causing our reduction to fail.

This is in direct contrast to the time-lock puzzle construction of Bitansky et al. [BGJ+16]: key to their adversary is amplification à la $\mathsf{BPP} \subset \mathsf{P}/\operatorname{poly}$. Note this is a non-uniform amplification, as it requires an advice string (i.e., the required random string). However, this non-uniform parallel amplification does not increase the overall depth of the computation, thus preserving their reduction. Such an amplification in our setting would additionally incur a $\operatorname{poly}(\lambda)$ factor in the aAT complexity of our adversary. Thus while our construction borrows heavily from [BGJ+16], there are many subtle differences between the two that require careful attention.

4.2 Minimality of Definition 4.5

We demonstrate that our definition of a memory-hard language is (essentially) minimal. In particular, given a memory-hard puzzle (as per Definition 4.3) with a uniformly succinct solving algorithm, we construct a memory-hard language (as per Definition 4.5).

Proposition 4.12. *Let $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ be a (g, ε)-memory hard puzzle such that for every security parameter λ and every positive difficulty parameter $t := t(\lambda)$, $\mathsf{Puz.Sol}$ is computable by some uniformly succinct circuit family $\{C_{t,\lambda}\}_{t,\lambda}$ of size $|C_{t,\lambda}| \leqslant t \cdot \operatorname{poly}(\lambda, \log(t))$. Define language $\mathcal{L}_{\mathsf{Puz}}$ as*

$$\mathcal{L}_{\mathsf{Puz}} := \{(Z, s) \colon s = \mathsf{Puz.Sol}(Z)\}.$$

Then $\mathcal{L}_{\mathsf{Puz}} \in \mathsf{SC}_t$ and is a (g, ε)-memory hard language.

Proof. Fix a difficulty parameter t. First note that given (Z, s), computing $s' = \mathsf{Puz.Sol}(Z)$ and checking $s = s'$ decides the language. Furthermore, by assumption $\mathsf{Puz.Sol}$ is uniformly succinct with circuit family $\{C_{t,\lambda}\}$ such that $|C_{t,\lambda}| \leqslant t \cdot \operatorname{poly}(\lambda, \log(t))$, which implies that $\mathcal{L}_{\mathsf{Puz}} \in \mathsf{SC}_t$. Second, if there exists a PRAM algorithm $\mathcal{A}$ which decides $\mathcal{L}_{\mathsf{Puz}}$ with advantage at least ε and $\mathsf{aAT}(\mathcal{A}) < g$, then we can easily construct adversary $\mathcal{A}'$ to violate the security of the memory-hard puzzle. Upon receiving (Z_b, Z_{1-b}, s_0, s_1) as specified by the security requirement of the memory-hard puzzle, $\mathcal{A}'$ obtains $b' \leftarrow \mathcal{A}(Z_b, s_0)$ and outputs $1 - b'$. Note that by construction $\mathsf{aAT}(\mathcal{A}') < g$. Furthermore, if $b = 0$ (resp., $b = 1$), then $(Z_0, s_0) \in \mathcal{L}_{\mathsf{Puz}}$ (resp., $(Z_0, s_0) \notin \mathcal{L}_{\mathsf{Puz}}$), and $\mathcal{A}$ outputs the correct answer $b' = 1$ (resp., $b' = 0$) with advantage at least ε. Thus $\mathcal{A}'$ outputs $(1 - b') = b$ with advantage at least ε, breaking the security of the memory-hard puzzle. □

We remark that the assumption that $\mathsf{Puz.Sol}$ is computable by some uniformly succinct circuit family is a modest assumption that is satisfied by our construction. In particular, we note that the succinct randomized encoding algorithm $\mathsf{sRE.Dec}$ of Garg and Srinivasan [GS18] used in our construction of $\mathsf{Puz.Sol}$ satisfies our requirement of uniform succinctness.

It is an interesting open question to determine whether or not memory-hard puzzles yield memory-hard languages unconditionally. Prior works have developed RAM to circuit transformations which

[7] Relaxing the definition of weakly memory-hard language to $1/\operatorname{poly}(\lambda)$ advantage instead of constant advantage removes the $\Theta(1/\varepsilon)$ factor.

transform RAM algorithms running in time t to circuits of size $t \cdot \text{polylog}(t)$ [BCGT13, BTVW14]. However, it is unclear if such transformations yield uniformly succinct circuits. Giving a uniformly succinct transformation would (nearly) resolve the question: if one of the many RAM to circuit transformations yielded a uniformly succinct circuit, then Proposition 4.12 holds unconditionally for any polynomial t.

5 One-time Memory-Hard Functions in the Standard Model

In this section we use memory-hard puzzles to construct (one-time) memory hard functions. Specifically, we present a construction of a one-time memory-hard function assuming memory-hard puzzles exist, the existence of puncturable psuedorandom functions (PPRFs), and the existence of indistinguishability obfuscation ($i\mathcal{O}$) for circuits. In fact, we conjecture that our construction is a secure multi-time MHF though we are unable to formally prove this for technical reasons. To the best of our knowledge, ours is the first construction of a memory-hard function in the standard model, assuming the existence of a suitably memory-hard language.

We first formally define one-time memory-hard functions and their security in the standard model. Prior definitions of memory-hard functions have been in the parallel random-oracle model (cf., [AS15, AT17]).

Definition 5.1 (One-Time Memory Hard Functions)**.** *A memory-hard function contains a pair of algorithms* ($\text{MHF}.\mathsf{Setup}, \text{MHF}.\mathsf{Setup}$) *which are descried as follows.*

- $\text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))$ *is a randomized algorithm that on input λ the security parameter and $t(\lambda)$ the hardness parameter, outputs public parameters* pp.
- $\text{MHF}.\mathsf{Eval}(\mathsf{pp}, x)$ *is a deterministic algorithm that on input the public parameter* pp *and message* $x \in \{0,1\}^\lambda$ *outputs* $h \in \{0,1\}^\lambda$.

We say that ($\text{MHF}.\mathsf{Setup}, \text{MHF}.\mathsf{Eval}$) *is a* one-time memory hard function *if the following hold*

Efficiency. $\text{MHF}.\mathsf{Eval}$ *is computable in time $t(\lambda) \cdot \text{poly}(\lambda)$ by a sequential RAM;*

Correctness. *There exists a negligible function μ such that for all x and all* $\mathsf{pp} \leftarrow \text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))$, *we have* $\Pr[h = h'] \geqslant 1-\mu(\lambda)$ *where* $h := \text{MHF}.\mathsf{Eval}(\mathsf{pp}, x)$ *and* $h' := \text{MHF}.\mathsf{Eval}(\mathsf{pp}, x)$ *(if $\mu(\lambda) = 0$ we say that the* MHF *is* perfectly correct*); and*

One-Time Memory Hard. *Given a function $g(\cdot,\cdot)$ we say that our construction is g-memory hard if there exists a polynomial t' such that for all polynomials $t(\lambda) > t'(\lambda)$ and every adversary A with area-time complexity* $\mathsf{aAT}(A) < y$ *for the function $y(\lambda) := g(t(\lambda), \lambda)$, there exists a negligible function $\mu(\lambda)$ such that for all $\lambda \in \mathbb{N}$ and every input $x \in \{0,1\}^\lambda$ we have*

$$\left| \Pr\left[A(x, h_b, \mathsf{pp}) = b\right] - \frac{1}{2} \right| \leqslant \mu(\lambda),$$

where the probability is taken over $\mathsf{pp} \leftarrow \text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))$, $b \xleftarrow{\$} \{0,1\}$, $h_0 \leftarrow \text{MHF}.\mathsf{Eval}(x, \mathsf{pp})$ *and a uniformly random string* $h_1 \xleftarrow{\$} \{0,1\}^\lambda$.

We are also interested in concretely secure one-time memory-hard functions.

Definition 5.2 (One-time (g, ε)-MHF)**.** *A* memory hard function $\text{MHF} = (\text{MHF}.\mathsf{Setup}, \text{MHF}.\mathsf{Eval})$ *is a* one-time (g, ε)-MHF *if there exists a polynomial t' such that for all polynomials $t(\lambda) > t'(\lambda)$ and* *every adversary A with area-time complexity* $\mathsf{aAT}(A) < y$*, where* $y(\lambda) = g(t(\lambda), \lambda)$*, and for all* $\lambda > 0$ *and* $x \in \{0,1\}^\lambda$ *we have*

$$\left| \Pr\left[A(x, h_b, \mathsf{pp}) = b\right] - \frac{1}{2} \right| \leqslant \varepsilon(\lambda),$$

where the probability is taken over $\mathsf{pp} \leftarrow \mathrm{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))$*,* $b \xleftarrow{\$} \{0,1\}$*,* $h_0 \leftarrow \mathrm{MHF}.\mathsf{Eval}(x, \mathsf{pp})$ *and a uniformly random string* $h_1 \in \{0,1\}^\lambda$*.*

5.1 Memory-Hard Function Construction

We construct a one-time memory-hard function from memory-hard puzzles, indistinguishable obfuscation ($i\mathcal{O}$), and (puncturable) psuedorandom functions (PPRFs). Our construction is shown in Construction 5.3, and we show that it is a one-time memory-hard function in Theorem 5.4.

Construction 5.3. *Let $i\mathcal{O}$ be an indistinguishablity obfuscator. Let $\lambda \in \mathbb{N}$ be the security parameter, let t be a polynomial in λ, let $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ be a PPRF, and let* $(\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *be a (g, ε)-memory-hard puzzle. We describe algorithms* $\mathrm{MHF}.\mathsf{Setup}$ *and* $\mathrm{MHF}.\mathsf{Eval}$ *in Figure 1.*

$\mathsf{pp} \leftarrow \mathrm{MHF}.\mathsf{Setup}(1^\lambda, t := t(\lambda))$

1. `Sample keys` $K_i \xleftarrow{\$} \{0,1\}^\lambda$ `for` $i \in [3]$
2. `Output` $\mathsf{pp} := i\mathcal{O}(\mathsf{prog}[K_1, K_2, K_3, \lambda, t])$

$h = \mathrm{MHF}.\mathsf{Eval}(\mathsf{pp}, x)$

1. `Compute` $Z \leftarrow \mathsf{pp}(x, \varnothing)$ $//Z = \mathsf{Puz.Gen}(1^\lambda, t, F(K_1, x); F(K_2, x))$
2. `Compute` $r' \leftarrow \mathsf{Puz.Sol}(Z)$
3. `Compute` $h \leftarrow \mathsf{pp}(x, r')$ $//h = F(K_3, x)$
4. `return` h

$\mathsf{prog}[K_1, K_2, K_3, \lambda, t](x, s')$

Internal (hardcoded) state: the set of secret PRF keys K_1, K_2, K_3, and hardness parameter λ and $t = t(\lambda)$.

1. `Compute` $s := F(K_1, x)$ `and` $r := F(K_2, x)$
2. `if` $s' = \varnothing$,
   - `return` $Z := \mathsf{Puz.Gen}(1^\lambda, t, s; r)$
3. `else if` $s = s'$, `return` $h = F(K_3, x)$
4. `else return` $\perp$

Figure 1: $\mathrm{MHF}.\mathsf{Setup}$, $\mathrm{MHF}.\mathsf{Eval}$, and prog.

Theorem 5.4. *Let t be a polynomial and let g be a function. If there exists a $(t_{\mathrm{PPRF}}, \varepsilon_{\mathrm{PPRF}})$-secure PPRF family, a $(t_{i\mathcal{O}}, \varepsilon_{i\mathcal{O}})$-secure $i\mathcal{O}$ scheme, and a $(g, \varepsilon_{\mathrm{MHP}})$-memory hard puzzle for $g(t(\lambda), \lambda) \leqslant \min\{t_{\mathrm{PPRF}}(\lambda), t_{i\mathcal{O}}(\lambda)\}$, then Construction 5.3 is a one-time $(g', \varepsilon_{\mathrm{MHF}})$-MHF for*

$$g'(t(\lambda), \lambda) = g(t(\lambda), \lambda)/p(\log(t(\lambda)), \lambda)^2,$$

where $\varepsilon_{\mathrm{MHF}}(\lambda) = 2 \cdot \varepsilon_{\mathrm{MHP}}(\lambda) + 3 \cdot \varepsilon_{\mathrm{PPRF}}(\lambda) + \varepsilon_{i\mathcal{O}}(\lambda)$ and $p(\log(t), \lambda)$ is a fixed polynomial which depends on the efficiency of underlying memory-hard puzzle and $i\mathcal{O}$.

To get a handle on Theorem 5.4, consider the following parameter settings. Let $\theta > 0$ be a small constant and suppose that t is suitably large such that $p(\log(t), \lambda)^2 = \Theta(t^c)$ for some suitably small constant $0 < c < \theta$. Then for $g(t, \lambda) = t^{2-\theta+c}$, $\varepsilon_{\mathrm{MHP}} = 1/(6\lambda^2)$, $\varepsilon_{\mathrm{PPRF}} = 1/(9\lambda^2)$, and $\varepsilon_{i\mathcal{O}} = 1/(3\lambda^2)$, our theorem yields a $(g', \varepsilon_{\mathrm{MHF}})$ for $g'(t, \lambda) = \Theta(t^{2-\theta})$ and $\varepsilon_{\mathrm{MHF}} = 1/\lambda^2$. Note that the exact parameters of the constructed MHF depend explicitly on the parameters of the underlying primitives used in the construction.

Efficiency of Construction 5.3. The efficiency of MHF.Eval follows directly from the run-time of prog and $\mathsf{Puz.Sol}$. Since $(\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ is a puzzle, we have that $\mathsf{Puz.Sol}$ runs in time $t(\lambda) \cdot \mathrm{poly}(\lambda)$. Next, the run-time of prog depends on the run-time of the PRF scheme and $\mathsf{Puz.Gen}$. In particular, PRFs are efficiently computable in time $\mathrm{poly}(\lambda)$ and $\mathsf{Puz.Gen}$ is computable in time $\mathrm{poly}(\lambda, \log(t(\lambda)))$. Therefore the efficiency of MHF.Eval is $t(\lambda) \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, \log(t(\lambda))) = t(\lambda) \cdot \mathrm{poly}(\lambda)$ as desired.

Correctness of Construction 5.3. Completeness of $(\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ guarantees that for every $\lambda \in \mathbb{N}$, $t < 2^\lambda$, $s \in \{0,1\}^\lambda$, and $Z \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s)$, we have that $s = \mathsf{Puz.Sol}(Z)$ with probability 1. This implies that for a fixed random string $r \in \{0,1\}^\lambda$, we have $s = \mathsf{Puz.Sol}(\mathsf{Puz.Gen}(1^\lambda, t, s; r))$. Once $\mathsf{pp} \leftarrow$ MHF.$\mathsf{Setup}(1^\lambda, t(\lambda))$ has been fixed, the PPRF keys are fixed within prog. This implies that on any input x, if $h_1, h_2 \leftarrow$ MHF.$\mathsf{Eval}(\mathsf{pp}, x)$ then $h_1 = h_2$ with probability 1.

One-Time Memory-Hardness of Construction 5.3. We give a high-level overview of proof of memory-hardness of our construction. The formal proof is presented to Appendix D. To prove memory-hardness, we transform a MHF attacker $\mathcal{A}$ with depth d and total size G (gates) into a MHP attacker $\mathcal{B}$ with depth $d' = d + p(\log(t), \lambda)/4$ and size $G' = G + p(\log(t), \lambda)/4$, leading to the multiplicative loss in aAT complexity.

To prove Theorem 5.4, we show how to use an MHF attacker $\mathcal{A}$ to break security of the underlying MHP. Our reduction involves four hybrids. Most of the security reduction is fairly standard. In the first hybrid H_0, we construct our memory-hard function as per Construction 5.3. Our second hybrid H_1 then modifies the construction by first puncturing the PPRF keys $K_i\{x_0, x_1\}$ at a target points x_0, x_1 and hardcode the values $s_j = F(K_1, x_j)$, $Z_j = \mathsf{Puz.Gen}(1^\lambda, t, s_j, F(K_2, x_j))$ and $h = F(K_3, x_j)$ for $j \in \{0,1\}$ to obtain a new (equivalent) program prog_1, relying on $i\mathcal{O}$ security for indistinguishability with the first hybrid H_0. In the third hybrid H_2 we modify s_0, s_1, Z_0, Z_1 and h_0, h_1 appropriately and rely on PPRF security for indistinguishability between H_2 and H_1. The most interesting step in our reduction is the final hybrid H_3 where we flip a bit b' and swap the puzzles Z_0, Z_1 if and only if $b' = 1$; i.e., we hardcode puzzles $Z_0' = Z_{b'}, Z_1' = Z_{1-b'}$. We rely on the security of the memory-hard puzzle to show that any attacker with low aAT complexity cannot distinguish between the last two hybrids.

Barriers to Proving Multi-Time Security. While we conjecture that our MHF construction achieves stronger multi-time security, we are unable to formally prove this. An interesting aspect of our final hybrid is that indistinguishability does not necessarily hold against an attacker with higher aAT complexity who could trivially distinguish between (s_0, s_1, Z_0, Z_1) and (s_0, s_1, Z_1, Z_0) by solving the puzzles Z_0 and Z_1. However, once the aAT complexity of the attacker is high enough to solve one puzzle, then we cannot rely on the MHP security for the indistinguishability of the final two hybrids. Proving multi-time security would involve proving that any attacker solving m distinct puzzles has aAT complexity that scales linearly in the number of puzzles i.e., any attacker with aAT complexity $o\left(m \cdot g(t(\lambda)\right)$ will fail to solve all m puzzles. In particular, while we expect that aAT complexity of the attacker is too small to solve all m puzzles the aAT complexity will still be large enough to solve *at least* one puzzle and distinguish between the hybrids that we defined in our security reduction.

Remark 5.5. For some MHF applications it is desirable to ensure that the evaluation algorithm is *data-independent*; i.e., the induced memory access pattern is independent of the input. Data-independent memory-hard functions (iMHFs) (and *computationally data-independent* memory-hard functions (ciMHFs) [ABZ20]) provide natural resistance to side-channnel attacks. We observe that Construction 5.3 is (computationally) data-independent as long as the underling $i\mathcal{O}$ and sRE schemes have data-independent evaluation algorithms, and that any candidate $i\mathcal{O}/\mathsf{sRE}$ scheme would satisfy this condition.

6 Locally Decodable Codes for Resource-Bounded Channels in the Standard Model

Our second application of memory-hard puzzles is constructing locally decodable codes (LDCs) for resource-bounded channels in the standard model. In the classical LDC setting the sender first encodes message $x \in \{0,1\}^k$ to obtain a longer $C = \mathsf{Enc}(x) \in \{0,1\}^K$ and transmits C over a noisy (adversarial) channel. The adversarial channel may flip up to δK bits in C before delivering the corrupted codeword $\tilde{C} \in \{0,1\}^K$ to the receiver. If the receiver wants to decode a bit x_i of the original message, the receiver can run a probabilistic local decoding procedure which will (with high probability) recover the correct value of x_i after examining at most ℓ bits of the codeword $\tilde{C}$. It is desirable to ensure that the code has constant rate ($K = \Theta(k)$) and small locality, e.g., $\ell = \mathrm{polylog}(k)$. Unfortunately, in the classical setting there are no known LDC constructions which achieve both of these properties. Blocki, Kulkarni, and Zhou [BKZ20] studied LDCs in the setting where the adversarial channel is resource-bounded and showed how to achieve LDCs with constant rate and locality $\ell = \mathrm{polylog}(k)$. In this setting, the channel is given C and is still allowed to flip up to δK bits, but the computations that the channel may perform are constrained in some way: e.g., space, aAT, sequential time. Arguably, any channel arising in nature can be modeled as a resource-bounded channel. In this sense, the constructions of Blocki, Kulkarni, and Zhou [BKZ20] would be suitable solution for communication over noisy channels arising in nature. However, the constructions of Blocki, Kulkarni, and Zhou [BKZ20] utilize random oracles, and the question of constructing resource-bounded LDCs in the standard model was left as an open question.

We show how to utilize cryptographic puzzles to construct resource-bounded LDCs in the standard model with constant rate and locality $\ell = \mathrm{polylog}(k)$. Our construction extends ideas of Blocki, Kulkarni, and Zhou [BKZ20] by replacing random oracles with cryptographic puzzles that are unsolvable by the resource-bounded channel in consideration. Combined with our construction of Memory-Hard Puzzles and the Time-Lock Puzzles from [BGJ+16], this yields concrete constructions of an LDC against aAT bounded channels and sequentially bounded channels, respectively. We additionally leverage a recent result of Block and Blocki [BB21] to obtain LDCs in the setting where the resource-bounded may insert/delete up to δK bits by using the "Hamming-to-InsDel" compiler of Block et al. [BBG+20] to transform our construction into a resource-bounded LDC for insertion-deletion errors.

6.1 Building Blocks

We begin by introducing definitions and building blocks relevant to our construction. For ease of presentation, we write this section assuming a binary alphabet $\{0,1\}$, but note that the definitions extend to any q-ary alphabet Σ.

Definition 6.1. *A (K,k)-coding scheme $C[K,k] = (\mathsf{Enc}, \mathsf{Dec})$ is a pair of algorithms $\mathsf{Enc}\colon \{0,1\}^k \to \{0,1\}^K$ and $\mathsf{Dec}\colon \{0,1\}^K \to \{0,1\}^k$. The rate of the scheme is defined as $R = k/K$.*

For two strings $x, y \in \{0,1\}^n$, we let HAM denote the *Hamming distance* between x and y, where $\mathsf{HAM}(x,y) := |\{i\colon x_i \neq y_i\}|$.

Definition 6.2. *A (K,k)-coding scheme $C[K,k] = (\mathsf{Enc}, \mathsf{Dec})$ is an (ℓ, δ, p)*-locally decodable code *(LDC) if Dec on input index $i \in [k]$ and oracle access to string y' such that $\mathsf{HAM}(\mathsf{Enc}(x), y') \leqslant \delta K$ outputs x_i with probability at least p, making at most ℓ queries to y'.*

The following definition is a slight variation of LDCs called LDC^*. An LDC^* is an LDC that is required to decode the entire original message while making as few queries as possible to its provided oracle.

Definition 6.3 ([BKZ20]). *A (K, k)-coding scheme $C[K, k] = (\mathsf{Enc}, \mathsf{Dec})$ is an (ℓ, δ, p)-LDC^* if Dec, with oracle access to a word y' such that $\mathsf{HAM}(\mathsf{Enc}(x), y') \leqslant \delta K$, makes at most ℓ queries to y' and outputs x with probability at least p.*

Using a repetition code one can construct a LDC^* where $\ell = \tilde{\Theta}(k)$ and $K \gg \ell, k$ can be as large as we want [BKZ20].

We also define *private* LDC*s* which are secure with respect to a particular class of algorithms $\mathbb{C}$. In this setting, the encoder/decoder share a secret key $\mathsf{sk} \in \{0,1\}^*$ which is not given to the channel. Fixing $\mathbb{C}$ as the class of all probabilistic polynomial time algorithms, Ostrovsky et al. [OPS07] constructed a private LDC with constant rate and locality polylog(k). We follow the definition of private LDCs from [BKZ20, BB21], which is equivalent to the definition from [OPS07].

Definition 6.4 (One-Time Private Key LDC). *A triple of probabilistic algorithms $C[K, k, \lambda] = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is $(\ell, \delta, p, \varepsilon, \mathbb{C})$-*private locally decodable code *(private LDC) against the class of algorithms $\mathbb{C}$ if*

1. *$\mathsf{Gen}(1^\lambda)$ is the key generation algorithm that takes as input 1^λ and outputs secret key $\mathsf{sk} \in \{0,1\}^*$ for security parameter λ;*
2. *$\mathsf{Enc}\colon \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^K$ is the encoding algorithm that takes as input message $x \in \{0,1\}^k$ and secret key sk and outputs a codeword $y \in \{0,1\}^K$;*
3. *$\mathsf{Dec}^{y'}\colon [k] \times \{0,1\}^* \to \{0,1\}$ is the decoding algorithm that takes as input index $i \in [k]$ and secret key sk, is additionally given query access to a corrupted codeword $y' \in \{0,1\}^{K'}$, and outputs $b \in \{0,1\}$ after making at most ℓ queries to y'; and*
4. *For all algorithms $\mathcal{A} \in \mathbb{C}$ and all messages $x \in \{0,1\}^k$ we have*

$$\Pr[\texttt{priv-LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p) = 1] \leqslant \varepsilon,$$

where the probability is taken over the random coins of $\mathcal{A}$ and Gen, and `priv-LDC-Sec-Game` *defined in Figure 2.*

`priv-LDC-Sec-Game`$(\mathcal{A}, x, \lambda, \delta, p)$:

1. The challenger generates a secret key $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, computes the codeword $y \leftarrow \mathsf{Enc}_{\mathsf{sk}}(x, \lambda)$ for the message x and sends the codeword y to the attacker.
2. The attacker outputs a corrupted codeword $y' \leftarrow \mathcal{A}(x, y, \lambda, \delta, p, k, K)$ where $y' \in \{0,1\}^K$ should have Hamming distance at most δK from y.
3. The output of the experiment is determined as follows:

$$\texttt{priv-LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p) = \begin{cases} 1 & \text{if } \mathsf{HAM}(y, y') \leqslant \delta K \text{ and } \exists i \in [k] \text{ s.t. } \Pr[\mathsf{Dec}_{\mathsf{sk}}^{y'}(i, \lambda) = x_i] < p \\ 0 & \text{otherwise} \end{cases}$$

If the output of the experiment is 1 (resp. 0), the attacker $\mathcal{A}$ is said to *win* (resp. *lose*) against C.

Figure 2: Definition of `priv-LDC-Sec-Game`, which defines the security of the a one-time private Hamming LDC against the class $\mathbb{C}$ of algorithms.

6.2 LDC Construction

Our construction is a general compiler which takes a private LDC, a LDC*, and a puzzle Puz which is hard for some class of algorithms $\mathbb{C}$ and outputs an LDC which is secure against the class of algorithms $\mathbb{C}$. We first formally define puzzles which are hard for algorithm class $\mathbb{C}$ (generalizing Definition 4.3) and then define LDCs which are secure against the class $\mathbb{C}$.

Definition 6.5 (($\mathbb{C}, \varepsilon$)-hard Puzzle)**.** *A puzzle* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *is a* $(\mathbb{C}, \varepsilon)$*-hard puzzle for algorithm class* $\mathbb{C}$ *there exists a polynomial* t' *such that for all polynomials* $t > t'$ *and every algorithm* $\mathcal{A} \in \mathbb{C}$*, there exists* λ_0 *such that for all* $\lambda > \lambda_0$ *and every* $s_0, s_1 \in \{0,1\}^\lambda$ *we have*

$$\left| \Pr[\mathcal{A}(Z_b, Z_{1-b}, s_0, s_1)] - \frac{1}{2} \right| \leqslant \varepsilon(\lambda),$$

where the probability is taken over $b \xleftarrow{\$} \{0,1\}$ *and* $Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ *for* $i \in \{0,1\}$*.*

Definition 6.6 ($\mathbb{C}$-Secure LDC)**.** *Let* $\mathbb{C}$ *be a class of algorithms. A* $(K,k)_q$ *coding scheme* $C[K,k]$ *is an* $(\ell, \delta, p, \varepsilon, \mathbb{C})$-locally decodable code *if*

1. $\mathsf{Enc}\colon \{0,1\}^k \to \{0,1\}^K$ *is the encoding algorithm that takes as input message* $x \in \{0,1\}^k$ *and outputs a codeword* $y \in \{0,1\}^K$*;*
2. $\mathsf{Dec}^{y'}\colon [k] \to \{0,1\}$ *is the decoding algorithm that takes as input index* $i \in [k]$*, is additionally given query access to a corrupted codeword* $y' \in \{0,1\}^{K'}$*, and outputs* $b \in \{0,1\}$ *after making at most* ℓ *queries to* y'*; and*
3. *For all algorithms* $\mathcal{A} \in \mathbb{C}$ *and all messages* $x \in \{0,1\}^k$ *we have*

$$\Pr[\texttt{LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p) = 1] \leqslant \varepsilon,$$

where the probability is taken over the random coins of $\mathcal{A}$ *and* `LDC-Sec-Game`*, defined in Figure 3.*

$\texttt{LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p)$:

1. The challenger computes $Y \leftarrow \mathsf{Enc}(x, \lambda)$ encoding the message x and sends $Y \in \{0,1\}^K$ to the attacker.
2. The channel $\mathcal{A}$ outputs a corrupted codeword $Y' \leftarrow \mathcal{A}(x, Y, \lambda, \delta, p, k, K)$ where $Y' \in \{0,1\}^K$ has Hamming distance at most δK from Y.
3. The output of the experiment is determined as follows:

$$\texttt{LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p) = \begin{cases} 1 & \text{if } \mathsf{HAM}(Y, Y') \leqslant \delta K \text{ and } \exists i \leqslant k \text{ s.t.} \Pr[\mathsf{Dec}^{Y'}(i, \lambda) = x_i] < p \\ 0 & \text{otherwise} \end{cases}$$

If the output of the experiment is 1 (resp. 0), the channel is said to *win* (resp. *lose*).

Figure 3: `LDC-Sec-Game` defining the interaction between an attacker and an honest party.

We now present our LDC construction.

Construction 6.7. *Let* $C_{\mathsf{p}}[K_{\mathsf{p}}, k_{\mathsf{p}}, \lambda] = (\mathsf{Gen}, \mathsf{Enc}_{\mathsf{p}}, \mathsf{Dec}_{\mathsf{p}})$ *be a private* LDC*, let* $C_*[K_*, k_*] = (\mathsf{Enc}_*, \mathsf{Dec}_*)$ *be a* LDC**, and let* $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ *be a* $(\mathbb{C}, \varepsilon')$*-hard puzzle. Let* t' *be the polynomial guaranteed by Definition 6.5. Then we construct* $C[K,k] = (\mathsf{Enc}, \mathsf{Dec})$ *as follows:*

$\mathsf{Enc}(x,\lambda)[C_{\mathsf{p}}, C_*, \mathsf{Puz}]$:	$\mathsf{Dec}^{Y'_{\mathsf{p}} \circ Y'_*}(i,\lambda)[C_{\mathsf{p}}, C_*, \mathsf{Puz}]$:
1. *Sample random seed* $s \xleftarrow{\$} \{0,1\}^{k_{\mathsf{p}}}$. 2. *Choose polynomial* $t > t'$ *and compute* $Z \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s)$, *where* $Z \in \{0,1\}^{k_*}$. 3. *Set* $Y_* \leftarrow \mathsf{Enc}_*(Z)$. 4. *Set* $\mathsf{sk} \leftarrow \mathsf{Gen}_{\mathsf{p}}(1^\lambda; s)$. 5. *Set* $Y_{\mathsf{p}} \leftarrow \mathsf{Enc}_{\mathsf{p}}(x, \lambda; \mathsf{sk})$. 6. *Output* $Y_{\mathsf{p}} \circ Y_*$.	1. *Decode* $Z \leftarrow \mathsf{Dec}_*^{Y'_*}$. 2. *Compute* $s \leftarrow \mathsf{Puz.Sol}(Z)$. 3. *Compute* $\mathsf{sk} \leftarrow \mathsf{Gen}_{\mathsf{p}}(1^\lambda; s)$. 4. *Output* $\mathsf{Dec}_{\mathsf{p}}^{Y'_{\mathsf{p}}}(i; \mathsf{sk})$.

We prove that if there exists a $\mathbb{C}$-hard puzzle, then Construction 6.7 is a $\mathbb{C}$-secure Hamming LDC.

Theorem 6.8. *Let $\mathbb{C}$ be a class of algorithms. Let $C_{\mathsf{p}}[K_{\mathsf{p}}, k_{\mathsf{p}}, \lambda]$ be a $(\ell_{\mathsf{p}}, \delta_{\mathsf{p}}, p_{\mathsf{p}}, \varepsilon_{\mathsf{p}})$-private* LDC *and let $C_*[K_*, k_*]$ be a (ℓ_*, δ_*, p_*)-*LDC**. Further assume that $\mathsf{Enc}_{\mathsf{p}}$, $\mathsf{Dec}_{\mathsf{p}}$, and Enc_* are contained in $\mathbb{C}$. If there exists a $(\mathbb{C}, \varepsilon')$-hard puzzle, then Construction 6.7 is a $(\ell, \delta, p, \varepsilon, \mathbb{C})$-locally decodable code $C[K,k] = (\mathsf{Enc}, \mathsf{Dec})$ with $k = k_{\mathsf{p}}$, $K = K_{\mathsf{p}} + K_*$, $\ell = \ell_{\mathsf{p}} + \ell_*$, $\delta = (1/K) \cdot \min\{\delta_* \cdot K_*, \delta_{\mathsf{p}} \cdot K_{\mathsf{p}}\}$, $p \geqslant 1 - k_{\mathsf{p}}(2 - p_{\mathsf{p}} - p_*)$, and $\varepsilon = k \cdot (\varepsilon_{\mathsf{p}} \cdot p + 2\varepsilon')/(1-p)$.*

As a direct corollary, if we assume the existence of a (g, ε')-memory hard puzzle then we directly obtain an LDC in the standard model which is secure against adversaries with low area-time complexity.

Corollary 6.9. *Let C_{p} be a private* LDC *and let C_* be a* LDC**. If there exists a (g, ε')-memory hard puzzle then Construction 6.7 is an $(\ell, \delta, p, \varepsilon, \mathbb{C})$-*LDC *against class*

$$\mathbb{C} := \{\mathcal{A}\colon\ \mathcal{A} \text{ is a PRAM algoirthm and } \mathsf{aAT}(\mathcal{A}) < g\}$$

for parameters $\varepsilon, \ell, \delta, p$, and ε defined in Theorem 6.8.

Efficiency. The efficiency of the scheme is directly given by the efficiency of C_{p}, C_*, and Puz. In particular, if all of the algorithms defined by $C_{\mathsf{p}}, C_*, \mathsf{Puz}$ are polynomial time, then Enc and Dec both run in polynomial time. We also remark that our LDC encoder Enc can be resource bounded: the encoder Enc only needs to be able to compute Puz.Gen, $\mathsf{Enc}_{\mathsf{p}}$, Enc_*, and $\mathsf{Gen}_{\mathsf{p}}$. Crucially, the encoder does not need to compute Puz.Sol. This is in contrast with [BKZ20], where their encoding function could not be resource-bounded, i.e., the construction inherently requires *both* the decoder and the encoder to evaluate a safe-function that cannot be computed by the resource-bounded channel.

Security. We formally show the security of our scheme in Appendix E and provide a high-level overview in this section. In the same vein as Blocki et al. [BKZ20], we employ the use of a *two-phased hybrid distinguisher*. To set up this distinguishing argument, first we consider two encoders Enc_0 and Enc_1. The encoder Enc_0 is exactly the encoder for our LDC in Construction 6.7. The encoder Enc_1 is the hybrid encoder and differs as follows: (1) Enc_1 is given additionally as input a secret key sk to be used with the private LDC C_{p}, rather than generating this key; and (2) the part of the codeword Y_* is constructed by sampling some s' independently and uncorrelated with sk, and then encoding $\mathsf{Enc}_*(\mathsf{Puz.Gen}(s'))$.

Given the encoders $\mathsf{Enc}_0, \mathsf{Enc}_1$, we construct our two-phase hybrid distinguisher $\mathcal{D} = (\mathcal{D}_1, \mathcal{D}_2)$ as follows. Phase one consists of the algorithm algorithm $\mathcal{D}_1$ which is given as input message x, and is additionally given access to both Enc_0 and Enc_1. Then $\mathcal{D}_1$ performs the following computations:

1. flips bit $b \xleftarrow{\$} \{0,1\}$; and

2. outputs codeword $Y_b \leftarrow \mathsf{Enc}_b(x)$.

The output of $\mathcal{D}_1(x)$ is then given to the adversarial channel, resulting in $Y'_b = Y'_{\mathsf{p},b} \circ Y'_* \leftarrow \mathcal{A}(Y_b)$ for $\mathcal{A} \in \mathbb{C}$. Here, $Y'_{\mathsf{p},b}$ is a substring of Y'_b that corresponds to the corruption of the codeword $Y_{\mathsf{p},b} \leftarrow \mathsf{Enc}_{\mathsf{p}}(x, \mathsf{sk}_b)$. Phase two consists of the algorithm $\mathcal{D}_2$ which is given as input the original message x, the secret key sk_b, and the corrupt codeword $Y'_{\mathsf{p},b}$, where b corresponds to the bit that was flipped when running $\mathcal{D}_1(x)$. Upon this input, $\mathcal{D}_2$ performs the following computations:

1. sample $i \xleftarrow{\$} [|x|]$;

2. run $x'_i \leftarrow \mathsf{Dec}_{\mathsf{p}}^{Y'_{\mathsf{p},b}}(i; \mathsf{sk}_b)$; and

3. output $b' = 0$ if $x_i \neq x'_i$; otherwise output $b' = 1$.

We say that the distinguisher $\mathcal{D}$ wins if $b = b'$.

Now if the adversary $\mathcal{A}$ is able to break `LDC-Sec-Game` with probability at least ε, we want to construct an algorithm $\mathcal{B} \in \mathbb{C}$ that uses this distinguishing argument to break the security of Puz. This is done as follows. Suppose $\mathcal{B}$ is given as input (Z_b, Z_{1-b}, s_0, s_1) for some $b \xleftarrow{\$} \{0,1\}$ that is unknown to $\mathcal{B}$ and where s_0, s_1 are uniformly random. Then $\mathcal{B}$ uses s_0 to generate sk, encodes $Y_* \leftarrow \mathsf{Enc}_*(Z_b)$, and encodes $Y_{\mathsf{p}} \leftarrow \mathsf{Enc}_{\mathsf{p}}(x, \mathsf{sk})$ for some fixed message x. We observe that if $b = 0$, then s_0 is the solution to $Z_0 = Z_b$, and thus Y_* is correlated with the secret key sk. Further, if $b = 1$, then s_0 is uncorrelated with $Z_b = Z_1$. Corrupted codeword $Y' \leftarrow \mathcal{A}(Y_{\mathsf{p}} \circ Y_*)$ is then obtained. Next, given x, secret key sk, and substring Y'_{p}, the algorithm simulates $\mathsf{Dec}_{\mathsf{p}}$ using sk and attempts to decode x_i for some arbitrary $i \in [|x|]$, obtaining x'_i. If $x'_i \neq x_i$, then $\mathcal{B}$ outputs $b' = 0$; otherwise it outputs $b' = 1$.

Now $\mathcal{B}$ is able to break the security of Puz as follows. If $b = 0$, then sk is correlated with Y_*. This implies that $\mathcal{A}$ is able to win `LDC-Sec-Game` with probability at least ε by assumption; in particular, it forces $\mathsf{Dec}_{\mathsf{p}}$ to output an incorrect bit for some index i with probability at least $(1-p)$, and the probability that the adversary selects this index is $1/k$. In this case, $b' = 0$ with probability at least $\varepsilon \cdot (1-p) \cdot (1/k)$. If $b = 1$, then sk is completely uncorrelated with Y_*, so information theoretically $\mathcal{A}$ cannot win `LDC-Sec-Game` except with probability at most ε_{p}. This implies that with probability at most $\varepsilon_{\mathsf{p}} \cdot p$ the decoder fails to output correctly on some index i, which implies that with probability at least $1 - \varepsilon_{\mathsf{p}} \cdot p$ the decoder outputs correctly on every bit. In this case, $b' = 1$ with probability at least $1 - \varepsilon_{\mathsf{p}} \cdot p$. This allows $\mathcal{B} \in \mathbb{C}$ to distinguish (Z_b, Z_{1-b}, s_0, s_1) with noticeable advantage $\Omega(\varepsilon \cdot (1-p) \cdot (1/k) - \varepsilon_{\mathsf{p}} \cdot p)$ thus breaking the security of the puzzle.

6.3 Resource-Bounded Locally Decodable Codes for Insertion-Deletion Errors in the Standard Model

Recently, Block and Blocki [BB21] proved that the so-called "Hamming-to-InsDel" compiler of Block et al. [BBG+20] extends to both the private Hamming LDC and resource-bounded Hamming LDC settings. That is, there exists a procedure which compiles any resource-bounded Hamming LDC to a resource-bounded LDC that is robust against *insertion-deletion errors* such that this compilation procedure preserves the underlying security of the Hamming LDC. We apply the result of Block and Blocki [BB21] to Construction 6.7 and obtain the first construction of resource-bounded locally decodable code for insertion-deletion errors in the standard model. We remark that the prior construction presented in [BB21] was in the random oracle model.

Corollary 6.10. *Let C_{p} be a private Hamming* LDC *and let C_* be a Hamming* LDC*. *If there exists a $(\mathbb{C}, \varepsilon')$-hard puzzle then there exists a $(\ell', \delta', p', \varepsilon'', \mathbb{C})$-*LDC *for insertion-deletion errors against class $\mathbb{C}$,* *where $\ell' = \ell \cdot O(\log^4(n))$, $\delta' = \Theta(\delta)$, $p' < p$, and $\varepsilon'' = \varepsilon/(1 - \mathsf{negl}(n))$. Here, n is the block length of the* LDC *resilient to insertion-deletion errors, and ℓ, δ, p, and ε are the parameters defined in Theorem 6.8.*

Corollary 6.11. *Let C_{p} be a private Hamming* LDC *and let C_* be a Hamming* LDC^*. *If there exists a (g, ε')-memory hard puzzle then there exists a $(\ell', \delta', p', \varepsilon'', \mathbb{C})$-*$\mathsf{LDC}$ *for insertion-deletion errors against class $\mathbb{C} = \{\mathcal{A}\colon\ \mathcal{A}$ is a PRAM algoirthm and $\mathsf{aAT}(\mathcal{A}) < g\}$, where $\ell' = \ell \cdot O(\log^4(n))$, $\delta' = \Theta(\delta)$, $p' < p$, and $\varepsilon'' = \varepsilon/(1 - \mathsf{negl}(n))$.*

7 Plausibility of Memory-Hard Languages

In this section we present evidence that memory-hard languages exist by giving a concrete example of a function that is computable by a succinctly describable circuit C_λ of size $t(\lambda) \cdot \mathrm{polylog}(t(\lambda))$ and which is provably memory hard when we assume that the underlying hash function is a random oracle. We remark that the succinct circuit describing C_λ can itself be constructed efficiently in time $\mathrm{poly}(\lambda, \log(t(\lambda)))$.

The rich line of work on the construction of memory-hard functions has (generally) followed this paradigm: given a depth-robust graph and random oracle H, a hash function $F_{G,H}$ corresponds to the final output of the following labeling function. Suppose $G = (V, E)$ and let $V = [N]$. For all $v \in V$, if $v = 1$ then we set the label of v as $L_v = H(x \circ 0^{(\lambda-1)\log(N)})$. Otherwise if $v > 1$ we set $L_v = H(L_{u_1}, L_{u_2}, \ldots, L_{u_k})$ where u_i is parent i of v.[8] The function $F_{G,H}(x)$ outputs L_N. Then $F_{G,H}$ is memory-hard based on the hardness of the underlying graph being *depth robust*. Briefly, a DAG $G = (V, E)$ is (e, d)-*depth robust* if after removing any $S \subset V$ nodes such that $|S| \leqslant e$, the remaining graph has depth at least d. As an example, consider the language $\mathcal{L} = \{(x, G)\colon \exists y \text{ s.t. } y = F_{G,H}(x)\}$. Then it is known that the language $\mathcal{L}$ is memory hard in the random oracle model [ACP+17] i.e., any algorithm evaluating $F_{G,H}$ in the parallel random oracle model has aAT cost at least $\Omega(ed\lambda)$. We interpret this as evidence that, under reasonable instantiations of the random oracle H, memory-hard languages exist under standard cryptographic assumptions; e.g., we can redefine the language as $\mathcal{L} = \{(x, G, \langle H\rangle)\colon \exists y \text{ s.t. } y = F_{G,H}(x)\}$ where $\langle H\rangle$ is the description of a hash function H such as SHA3 or the Argon2 round function [BDK16].

One key property needed by the function $F_{G,H}$ in order for the language $\mathcal{L}$ to be a memory-hard language is our uniform succinctness condition. In particular, we must have a function $F_{G,H}$ which is uniformly succinct, and such a function would need the graph G and the hash function H to be uniformly succinct. While certain concrete hash functions, such as SHA3, certainly have uniformly succinct description, many constructions of $F_{G,H}$ are only memory-hard when the underlying DAG G is constructed via a random process. Such a graph cannot ever hope to be uniformly succinct, and therefore many MHF constructions of the form $F_{G,H}$ would not yield a memory-hard language, even if H was uniformly succinct.

Remark 7.1. The discussion in this section is purely concerned with the plausibility of the existence of memory-hard languages. We stress that we do not know how to formally prove the existence of such languages, barring some major advances in circuit lower bounds.

7.1 Powers of Two Graph

We present a deterministic DAG G from folklore that is both uniformly succinct and provably depth-robust. This graph is known as the *powers of two graph*. Define a D $G_{\mathsf{Po2},N}$ on $|V| = N$ vertices as follows. Suppose $n = \log(N)$. For every $v \in V$, we define $\mathsf{parents}(v) := (u_0, \ldots, u_{n-1})$ where $u_i = v - 2^i$

[8] We remark that this is one general flavor of constructions of memory-hard functions. However, not all constructions follow this exact methodology. We state this methodology here for intuition and ease of presentation.

if $2^i < v$, and $u_i = 0 \notin V$ otherwise. Then in the random oracle model, the function $F_{G_{\mathsf{Po2},N},H}$ is a provably memory-hard function.

Note that by definition, there is an efficient uniform and deterministic algorithm A which on input N and a node $v \in \{0, 1, \ldots, N-1\}$ outputs the parents of v; thus $G_{\mathsf{Po2},N}$ is uniformly succinct. This is in contrast to other depth-robust graphs (e.g., [ABP17, ABH17, ABP18]) which use a random algorithm A to generate parents, which implies $N \cdot \operatorname{polylog}(N)$ random-bits are needed just to specify the graph G itself. Working with $G_{\mathsf{Po2},N}$ allows us to construct a function $F_{G_{\mathsf{Po2},N},H}$ that is uniformly succinct whenever H is uniformly succinct (which is again a reasonable assumption for many concrete hash functions such as SHA3).

We dedicate the remainder of this section to proving that the language $\mathcal{L}$ described at the start of this section is memory-hard given the function $F_{G_{\mathsf{Po2},N},H}$.

Lemma 7.2 ([per]). *The graph $G_{\mathsf{Po2},N}$ on N vertices is (e, d)-depth robust for $e = \Omega(N/\operatorname{polylog}(N))$ and $d = \Omega(N/\operatorname{polylog}(N))$.*

Lemma 7.2 is proved in Appendix F.1. The Po2 graph also has large area-time complexity. Applying a result of Alwen et al. [ABP17] we immediately have that the cumulative pebbling complexity of the graph G is at least $e \cdot d = \Omega(N^2/\operatorname{polylog}(N))$ and, therefore, the aAT complexity is at least $\Omega(N^2 \cdot \lambda/\operatorname{polylog}(N))$.

Lemma 7.3 ([ABP17]). *In the parallel random oracle model, the function $F^H_{G_{\mathsf{Po2},N}}$ has area-time complexity at least $\Omega(N^2 \cdot \lambda/\operatorname{polylog}(N))$.*

Let $\mathcal{R}_{\mathsf{Po2},N} = \left\{(x, y) \colon y = F^H_{G_{\mathsf{Po2},N}}(x)\right\}$ and let $\mathcal{L}_{\mathsf{Po2},N}$ be the language for relation $\mathcal{R}_{\mathsf{Po2},N}$.

Proposition 7.4. *Let $N, \lambda \in \mathbb{N}$. Let $\mathcal{L}^\lambda_{\mathsf{Po2},N}$ be the language for the relation $\mathcal{R}_{\mathsf{Po2},N}$ instantiated with $x, y \in \{0,1\}^\lambda$ and hash function $H_{N,\lambda} \colon \{0,1\}^{\lambda \log(N)} \to \{0,1\}^\lambda$ such that $H_{N,\lambda}$ is a uniformly succinct circuit of size $N \cdot \operatorname{poly}(\lambda, \log(N))$. Then $\mathcal{L}^\lambda_{\mathsf{Po2},N} \in \mathsf{SC}_{N'}$ for $N' = N^2$.*

The full proof is presented in Appendix F.2.

Remark 7.5. Real-world hash functions satisfy the requirements of Proposition 7.4. Moreover, the construction of Proposition 7.4 is easily extended to any (e, d)-depth robust graph that has a uniformly succinct circuit representation, albeit with different parameters.

8 Space Efficient Simulation of Single Tape Turing Machines

In this section, we prove that any single-tape Turing machine running in time $t := t(n)$ for inputs of size n is decidable by a PRAM algorithm with aAT complexity at most $O(t^{1.8} \cdot \log(t))$. We believe this result may be of independent interest. This shows that if we modify the language class SC_t (Definition 4.4) to require any language in this class to be decided by a time t single-tape Turing machine, then memory-hard languages can only be secure against adversaries with aAT complexity $o(t^{1.8} \cdot \log(t))$. We note that we don't prove that our simulation is optimal, thus it is possible to give a simulation with less aAT complexity. We dedicate this section to proving the following theorem.

Theorem 8.1. *For any language $\mathcal{L}$ decidable in time $t(n)$ by a single-tape Turing machine for inputs of size n, there exists a constant $c > 0$ such that $\mathcal{L}$ is decidable by a PRAM algorithm with* aAT *complexity at most $c \cdot t(n)^{1.8} \cdot \log(t(n))$.*

Note that Theorem 8.1 only holds for single-tape Turing machines. It is an interesting open question if there is a similar result for multi-tape Turing machines. In particular, if one could show such a simulation for two-tape Turing machines, then one can leverage the multi-tape to oblivious two-tape Turing machine reduction of [PF79] to obtain a similar result for multi-tape Turing machines; or one could prove a simulation directly.

8.1 Brief Review of Turing Machines

A single-tape Turing machine M consist of the three elements: (1) an *infinite tape* which includes cell numbered as $\mathbb{Z}^+$; (2) a two-way read/write *head* which is the program counter and indicates the current state of the machine; and (3) a finite set of *controlling states*, $Q = \{\eta_1, \cdots, \eta_m\}$ and a *transition* function δ. For each Turing machine M, we define the input alphabet as Σ, and the tape alphabet as $\Gamma \supseteq \Sigma \cup \{\square\}$ such that $\square \notin \Sigma$ is the blank symbol. Semantically, a Turing machine M works as follows:

- **Initial configuration:** The input $x_1, \ldots, x_n$ is initially placed in cells $1, \ldots, n$ and all other cells contain $\square$. In this configuration, the location of head is on the first cell of the tape and $\eta_{start} \in Q$ is the initial state of the machine.

- **Transition Details:** The transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ takes as input the current state $\eta \in Q$ of M along with the current cell contents $\sigma \in \Gamma$ and outputs a new state $\eta' \in Q$, updates the cell contents with $\sigma' \in \Gamma$ and moves the head left or right. We let $T[i, t'] \in \Gamma$ denote the content of cell i at time t'.

8.2 Simulation Overview

For any language $\mathcal{L}$ such that $\mathcal{L}_n$ is decidable in time $t(n)$ by a Turing machine, we show how to simulate this Turing machine via a PRAM algorithm in time t and space $O(t^{0.8} \cdot \log(t))$, and thus aAT complexity at most $O(t^{1.8} \cdot \log(t))$. To show this, let M be any Turing machine which halts after t steps on inputs of size n. Then we build our PRAM algorithm $\mathcal{A}$ which simulates M using space $c \cdot t^{0.8} \cdot \log(t)$ for some constant $c > 0$.

To begin we describe a simulator $\mathcal{A}'$ which uses space $O\left(t^{0.75} \cdot \log(t)\right)$, but requires a hint h_x that depends on the specific input x (i.e., $\mathcal{A}'$ is a non-uniform algorithm). Intuitively, the hint allows us to compress intervals on the TM tape in such a way that the contents of the tape can still be recovered in reasonable time. We can further utilize parallelism to ensure that our simulation is never delayed. We then show how to modify the simulator to eliminate the input dependent hint h_x. This modification increases our space usage slightly to $O\left(t^{0.8} \cdot \log(t)\right)$.

8.3 Simulation Details

We first define some notation we use throughout the remainder of this section.

- We let $T[i, t', M, x] \in \Gamma$ denote the content of cell i at time t' when Turing machine M is run on input x. Similarly, we let $S[t', M, x] \in Q$ denote the state of the Turing machine at time t'. When M and x are clear from context we simplify and write $T[i, t']$ and $S[t']$ respectively.

- $\chi(i, t', M, x)$ denotes the number of visits by the TM head at the i-th cell of M's tape up to time t'. When M and x are clear from context, we simply write $\chi(i, t')$.

- $\chi(i, j, t', M, x)$ denotes the total summation of visits by the TM head for all cells $\{i, i+1, \ldots, j\}$. So we have $\chi(i, j, t') = \sum_{k=1}^{j} \chi(k, t')$.We write $\chi(i, j, t')$ when M and x are clear from context.

- γ_1: We partition the TM tape into t/γ_1 intervals of size $O(\gamma_1)$.

- γ_2: We maintain the invariant that if our Turing machine head is on cell j at time t' then we also have $T[j - \gamma, t'], T[j - \gamma + 1, t'], \ldots, T[j, t'], T[j + 1, t'], \ldots, T[j + \gamma, t']$ stored in memory — the current contents of the Turing machine for any cell that we might visit within γ_2 steps.

Based on the above definitions, we have the following useful observation.

Observation 8.2. *For all times $t' \leqslant t$ and all pairs $i < j \leqslant t$ there exists $i \leqslant k \leqslant j$ such that $\chi(k, t') \leqslant \frac{\chi(i,j,t')}{j-i+1}$. In particular, if $\chi(i, j, t') \leqslant \gamma_2$ and $j - i + 1 \leqslant \gamma_1$ then $\chi(k, t') \leqslant \frac{\gamma_2}{\gamma_1}$.*

This observation follows immediately from the definition since $\frac{\chi(i,j,t')}{j-i+1}$ is the average value of $\chi(k, t')$ for $k \in [i, j]$.

Definition 8.3 (Compressed state)**.** *Given the Turing machine M, cell indices i, j of the tape and the current time t', we define* $\texttt{Compress}(i, j, t')$ *which is the following states:*

- $t^i_1 < t^i_2 < \ldots < t^i_a$ *and* $T[i, t^i_1], \ldots, T[i, t^i_a]$ *and* $S[t^i_1], \ldots, S[t^i_a]$ *where* $a = \chi(i, t')$.
- $t^j_1 < t^j_2 < \ldots < t^j_b$ *and* $T[j, t^j_1], \ldots, T[j, t^j_a]$ *and* $S[t^j_1], \ldots, S[t^j_a]$ *where* $b = \chi(j, t')$.

Here, t^i_k (resp. t^j_k) denotes the time of the i-th visit to cell i on the Turing Machine tape.

Lemma 8.4 (Decompression lemma)**.** *Given the compressed state information* $\texttt{Compress}(i, j, t')$ *for all visits to cells i and j, the current tape contents at time t' can be recovered for an arbitrary interval $[i, j]$ in time $\chi(i, j, t')$ with extra space usage $O(j - i + 1)$.*

The proof of Lemma 8.4 is deferred to Appendix G.

Lemma 8.5 (Recompression lemma)**.** *Given Turing machine M, tape indices i, j and the current time t', we can recover both the tape contents between i and j, and the value $\chi(k, t')$ such that $k \in [i, j]$ is associated to the lowest in total time $\chi(i, j, t')$ and extra space $O\left(\log(t') + \dfrac{\chi(i, j, t')}{j - i + 1}\right)$.*

The proof of Lemma 8.5 is deferred to Appendix G.

8.3.1 Warm-up Discussion

Before we prove Theorem 8.1, consider simulator $\mathcal{A}'(x)$ given hint h_x to simulate Turing machine M in time t and space $t^{3/4} \cdot \log(t)$. In particular, h_x encodes indices $i_1, \ldots, i_{t/\gamma_1}$ with $i_j \in [(j-1) \cdot \gamma_1, j \cdot \gamma_1]$ and bits $b_1, \ldots, b_{t/\gamma_1}$ such that $b_j = 1$ if and only if $\chi((j-1) \cdot \gamma_1 + 1, j \cdot \gamma_1, t) \leqslant \gamma_2$. Furthermore, for each j with $b_j = 1$ we can require that $\chi(i_j, t) \leqslant \gamma_2/\gamma_1$ by Observation 8.2. For each i_j and i_{j+1} with $b_j = b_{j+1} = 1$ the simulator will store state $\texttt{Compress}(i_j, i_{j+1}, t')$ and we call the interval $[i_j, i_{j+1}]$ compressible; otherwise, we call j incompressible. The simulator maintains the invariant that the contents of the Turning machine tape at locations $i - 4 \cdot \gamma_2$ to $i + 4 \cdot \gamma_2$ are always stored in memory. Furthermore, for any j with $b_j = 1$ we will maintain the invariant that the content of the Turing machine tape at all locations in the interval from i_{j-1} to i_{j+1} are stored in memory.

The crucial observation is that if $b_j, b_{j+1} = 1$, then based on Lemma 8.4 we can quickly, within $2 \cdot \gamma_2$ steps, recover the current contents of the Turing machine tape at all cells in the interval i_j to i_{j+1} using $\texttt{Compress}(i_j, i_{j+1})$.

For time complexity, we point out that based on selection of i_j (according to hint) the number of visits at cell i_j is bounded to γ_2/γ_1. Once the right starting point determined, we can recover the machine state and content of our intended cell by $2\gamma_2$ steps as based on Observation 8.2 we have $\chi(i_j, i_{j+1}, t') \leqslant \gamma_2$ which implies the worst case.

We can also do this in parallel for any value of j with $b_j, b_{j+1} = 1$ to maintain our invariant that we always keep the contents of the turning machine tape at locations $i - 4 \cdot \gamma_2$ to $i + 4 \cdot \gamma_2$ in memory. In particular, if $|i_j - i| \leqslant 6\gamma_2$ and $b_j = b_{j+1} = 1$ then we start the decompression process. If $b_j = 0$ or $b_{j+1} = 0$ then the contents of these $\leqslant 2\gamma_1$ cells are already stored. We have at most $2t/\gamma_2$ uncompressible intervals i.e., j s.t. $b_j = 0$ or $b_{j+1} = 0$. Thus, we require space *at most* $\gamma_1 \cdot 2t/\gamma_2$ to store these uncompressible intervals. We require space *at most* $\gamma_2/\gamma_1 \cdot O(\log(t))$ for each index i_j with

$b_j = 1$. Thus, we use total space $t/\gamma_1 \cdot \gamma_2/\gamma_1 \cdot O(\log(t))$ to store the compressible intervals. Finally, we have *at most* $6\gamma_2/\gamma_1$ intervals that are being decompressed at any point in time and we require additional space γ_1 for each such interval. The overall space usage is $O\left(\gamma_2 + \frac{t\gamma_1}{\gamma_2} + \frac{t\cdot\gamma_2\cdot\log(t)}{\gamma_1^2}\right)$. We can minimize by setting $\gamma_1 = \sqrt{t}$ and $\gamma_2 = t^{3/4}$ which gives us overall space usage $O(t^{3/4} \cdot \log(t))$. This gives that the aAT complexity of our PRAM algorithm is at most $O(t^{1.75} \cdot \log(t))$.

8.4 Proof of Theorem 8.1

The proof idea is similar to the way we designed the simulator $\mathcal{A}'(x)$. The main difference here is that the simulator does not have access to the hint. So we will show that there still exist a simulator like $\mathcal{A}(x)$ which reconstructs the removed cell contents with an extra space and the same order of running time. This extra space results in total $\mathsf{aAT}(\mathcal{A}, n) = c \cdot t(n)^{1.8} \cdot \log(t(n))$. We use Lemma 8.5 to prove this lemma.

Essentially, we need to extract the points $b_j, b_{j+1} = 1$ and use them to set $\texttt{Compress}(i_j, i_{j+1}, t')$ as in this case $\mathcal{A}(x)$ is not given the hint. Initially, we set $i_j = \gamma_1 j$ and set $b_j = 1$ for these potential points. Then we dynamically update these points to ensure that $\chi(i_j, t') \leqslant 2\gamma_2/\gamma_1$ i.e., $b_j = 1$; otherwise, we find a new point $i_{j'}$ and set $b_{j'} = 1$. For updating the point, we use the results of recompression lemma, i.e., Lemma 8.5, we can extract the $\chi(k, t')$ for all $k \in [i_{j-1}, i_{j+1}]$ and find indexes $i_j - \Delta \leqslant i'_j \leqslant i_j + \Delta$ and $i_{j+1} - \Delta \leqslant i'_{j+1} \leqslant i_{j+1} + \Delta$ which are corresponding to the minimum number of visits satisfying $\chi(i'_j, t'), \chi(i'_{j+1}, t') \leqslant 2\gamma_2/\Delta$. Here, as the size of interval is γ_1 we can set $\Delta = \alpha(i_{j+1} - i_j) = \alpha\gamma_i$. Without loss of generality we can consider the constant value $\alpha = \frac{1}{10}$ and we have $\Delta = \frac{\gamma_1}{10}$. Therefore, replacing the Δ in the bounds we will have $\chi(i'_j, t'), \chi(i'_{j+1}, t') \leqslant 20\gamma_2/\gamma_1$, which in fact the results we are looking for. Now, we just need to update $b_j = b_{j+1} = 0$ and set $b'_j = b'_{j+1} = 1$ which are actually the flags corresponding to i'_j and i'_{j+1}. As the last step, we also need to compute and store $\texttt{Compress}(i'_j, i'_{j+1}, t')$.

If $\chi(i_j, t'), \chi(i_{j+1}, t') > 2\gamma_2/\gamma_1$, and we need to find alternative indexes i'_j, i'_{j+1} then Lemma 8.5 implies that extra space cost which would be $O(\log(t) + \gamma_2/\gamma_1)$. As we have at most γ_2 such cases, thus, the total extra space usage is at most $O(\gamma_2 \cdot \log(t) + \gamma_2^2/\gamma_1)$ (in comparison with simulator $\mathcal{A}'(x)$ with hint). Therefore, the overall space usage is $O\left(\gamma_2 \cdot \log(t) + \frac{t\cdot\gamma_1}{\gamma_2} + \frac{t\cdot\gamma_2\cdot\log(t)}{\gamma_1^2} + \gamma_2^2/\gamma_1\right)$. Now we can minimize by setting $\gamma_2 = t^{3/5}$ and $\gamma_1 = t^{2/5}$ to achieve overall space usage $O\left(t^{4/5} \cdot \log(t)\right)$.

8.5 The Necessity of $\texttt{Compress}(i_j, i_{j+1}, t')$

Here we discuss why we need to store all visits information of cell i_j. If we only store the tuple associated with the first visit of head at location i_j (that is, tuple $(t_{i_j,1}, S[t_{i_j,1}], T[i, t_{i_j,1}])$), then it may take more time for the simulator to recover the cells of the target interval (more than γ_2 steps). This is due to the fact that the head may return to the starting cell i_j and then exit the interval $[i_j, i_{j+1}]$ and continue for a long period of time outside it. This imposes a delay in the final running time an we cannot recover the interval contents in time at most $2 \cdot \gamma_2$.

So for handling this problems, we need to store extra tuples for cell i_j regarding all visits. This is one scenario implies why we need to store tuples for all visits. In this case, when head decides to go out of the interval, we halt and start simulation from the next stored tuple $(t_{i_j,k}, S[t_{i_j,k}], T[i, t_{i_j,k}])$ for some $1 < k \leqslant a$ ($a = \chi(i_j, t')$) which guides the simulation inside the interval (the head continues to go to the right of i_j).

In addition, we cannot start at position corresponding to the last visit as we do not have the last visit information of the neighboring cell, i.e., the tuple $(t_{i_j+1,a'}, S[t_{i_j+1,a'}], T[i_j+1, t_{i_j+1,a'}])$ (which are basically blank; i.e., $\square$) so we may not be able to reconstruct the correct values. Therefore, considering both these scenarios, we need to store $\texttt{Compress}(i_j, i_{j+1})$ for all cells that $b_j = b_{j+1} = 1$.

Acknowledgments

Mohammad Hassan Ameri was supported in part by the National Science Foundation under award #1755708 and by IARPA under the HECTOR program and by a Summer Research Grant at Purdue University. Alexander R. Block was supported in part by the National Science Foundation under NSF CCF #1910659. Jeremiah Blocki was supported in part by the National Science Foundation under awards CNS #1755708, CNS #2047272, and CCF #1910659 and by IARPA under the HECTOR program.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[AB16] Joël Alwen and Jeremiah Blocki. Efficiently computing data-independent memory-hard functions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 241–271. Springer, Heidelberg, August 2016. doi:10.1007/978-3-662-53008-5_9.

[ABH17] Joël Alwen, Jeremiah Blocki, and Ben Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1001–1017. ACM Press, October / November 2017. doi:10.1145/3133956.3134031.

[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 3–32. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56617-7_1.

[ABP18] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 99–130. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_4.

[ABZ20] Mohammad Hassan Ameri, Jeremiah Blocki, and Samson Zhou. Computationally data-independent memory hard functions. In Thomas Vidick, editor, *ITCS 2020*, volume 151, pages 36:1–36:28. LIPIcs, January 2020. doi:10.4230/LIPIcs.ITCS.2020.36.

[ACK+16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 358–387. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49896-5_13.

[ACP+17] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 33–62. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56617-7_2.

[AIK04] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. In *45th FOCS*, pages 166–175. IEEE Computer Society Press, October 2004. doi:10.1109/FOCS.2004.20.

[AP20] Shweta Agrawal and Alice Pellet-Mary. Indistinguishability obfuscation without maps: Attacks and fixes for noisy linear FE. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 110–140. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45721-1_5.

[App17] Benny Applebaum. Garbled circuits as randomized encodings of functions: a primer. Cryptology ePrint Archive, Report 2017/385, 2017. https://eprint.iacr.org/2017/385.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 595–603. ACM Press, June 2015. doi:10.1145/2746539.2746622.

[AT17] Joël Alwen and Björn Tackmann. Moderately hard functions: Definition, instantiations, and applications. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 493–526. Springer, Heidelberg, November 2017. doi:10.1007/978-3-319-70500-2_17.

[BB21] Alexander R. Block and Jeremiah Blocki. Private and resource-bounded locally decodable codes for insertions and deletions. *CoRR*, abs/2103.14122, 2021. ISIT 2021, to appear. URL: https://arxiv.org/abs/2103.14122, arXiv:2103.14122.

[BBG+20] Alexander R. Block, Jeremiah Blocki, Elena Grigorescu, Shubhang Kulkarni, and Minshen Zhu. Locally decodable/correctable codes for insertions and deletions. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 182 of *LIPIcs*, pages 16:1–16:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSTTCS.2020.16.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 401–414. ACM, January 2013. doi:10.1145/2422436.2422481.

[BDGM20] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Candidate iO from homomorphic encryption schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 79–109. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45721-1_4.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[BGGZ19] Jeremiah Blocki, Venkata Gandikota, Elena Grigorescu, and Samson Zhou. Relaxed locally correctable codes in computationally bounded channels*. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 2414–2418, 2019. doi:10.1109/ISIT.2019.8849322.

[BGH+06] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM J. Comput.*, 36(4):889–974, 2006. A preliminary version appeared in the Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC).

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001. doi:10.1007/3-540-44647-8_1.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014. doi:10.1007/978-3-642-54631-0_29.

[BGJ+16] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In Madhu Sudan, editor, *ITCS 2016*, pages 345–356. ACM, January 2016. doi:10.1145/2840728.2840745.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 439–448. ACM Press, June 2015. doi:10.1145/2746539.2746574.

[BGT14] Nir Bitansky, Sanjam Garg, and Sidharth Telang. Succinct randomized encodings and their applications. Cryptology ePrint Archive, Report 2014/771, 2014. https://eprint.iacr.org/2014/771.

[BGZ18] J. Brakensiek, V. Guruswami, and S. Zbarsky. Efficient low-redundancy codes for correcting multiple deletions. *IEEE Transactions on Information Theory*, 64(5):3403–3410, 2018. doi:10.1109/TIT.2017.2746566.

[BHK+19] Jeremiah Blocki, Benjamin Harsha, Siteng Kang, Seunghoon Lee, Lu Xing, and Samson Zhou. Data-independent memory hard functions: New attacks and stronger constructions. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 573–607. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26951-7_20.

[BIJ+20] James Bartusek, Yuval Ishai, Aayush Jain, Fermi Ma, Amit Sahai, and Mark Zhandry. Affine determinant programs: A framework for obfuscation and witness encryption. In Thomas Vidick, editor, *ITCS 2020*, volume 151, pages 82:1–82:39. LIPIcs, January 2020. doi:10.4230/LIPIcs.ITCS.2020.82.

[BKZ20] Jeremiah Blocki, Shubhang Kulkarni, and Samson Zhou. On locally decodable codes in resource bounded channels. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 16:1–16:23. Schloss Dagstuhl, June 2020. doi:10.4230/LIPIcs.ITC.2020.16.

[BN00] Dan Boneh and Moni Naor. Timed commitments. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 236–254. Springer, Heidelberg, August 2000. doi:10.1007/3-540-44598-6_15.

[BRZ18] Jeremiah Blocki, Ling Ren, and Samson Zhou. Bandwidth-hard functions: Reductions and lower bounds. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1820–1836. ACM Press, October 2018. doi:10.1145/3243734.3243773.

[BTVW14] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. Cryptology ePrint Archive, Report 2014/846, 2014. https://eprint.iacr.org/2014/846.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013. doi:10.1007/978-3-642-42045-0_15.

[BZ17] Jeremiah Blocki and Samson Zhou. On the depth-robustness and cumulative pebbling cost of Argon2i. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 445–465. Springer, Heidelberg, November 2017. doi:10.1007/978-3-319-70500-2_15.

[CGHL20] Kuan Cheng, Venkatesan Guruswami, Bernhard Haeupler, and Xin Li. Efficient linear and affine codes for correcting insertions/deletions, 2020. arXiv:2007.09075.

[CHL+19] Kuan Cheng, Bernhard Haeupler, Xin Li, Amirbehshad Shahrasbi, and Ke Wu. Synchronization strings: Highly efficient deterministic constructions over small alphabets. In Timothy M. Chan, editor, *30th SODA*, pages 2185–2204. ACM-SIAM, January 2019. doi:10.1137/1.9781611975482.132.

[CJLW18] Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Deterministic document exchange protocols, and almost optimal binary codes for edit errors. In Mikkel Thorup, editor, *59th FOCS*, pages 200–211. IEEE Computer Society Press, October 2018. doi:10.1109/FOCS.2018.00028.

[CJLW19] Kuan Cheng, Zhengzhong Jin, Xin Li, and Ke Wu. Block edit errors with transpositions: Deterministic document exchange protocols and almost optimal binary codes. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *ICALP 2019*, volume 132 of *LIPIcs*, pages 37:1–37:15. Schloss Dagstuhl, July 2019. doi:10.4230/LIPIcs.ICALP.2019.37.

[CL20] Kuan Cheng and Xin Li. Efficient document exchange and error correcting codes with asymmetric information, 2020. arXiv:2007.00870.

[CT19] Binyi Chen and Stefano Tessaro. Memory-hard functions from cryptographic primitives. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 543–572. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26951-7_19.

[DGY10] Zeev Dvir, Parikshit Gopalan, and Sergey Yekhanin. Matching vector codes. In *51st FOCS*, pages 705–714. IEEE Computer Society Press, October 2010. doi:10.1109/FOCS.2010.73.

[DGY11] Zeev Dvir, Parikshit Gopalan, and Sergey Yekhanin. Matching vector codes. *SIAM J. Comput.*, 40(4):1154–1178, 2011.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Heidelberg, August 1993. doi:10.1007/3-540-48071-4_10.

[Efr09] Klim Efremenko. 3-query locally decodable codes of subexponential length. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 39–44. ACM Press, May / June 2009. doi:10.1145/1536414.1536422.

[Efr12] Klim Efremenko. 3-query locally decodable codes of subexponential length. *SIAM J. Comput.*, 41(6):1694–1703, 2012.

[EGS75] Paul Erdös, Ronald L. Graham, and Endre Szemerédi. On sparse graphs with dense long paths. Technical report, Stanford, CA, USA, 1975.

[FLW14] Christian Forler, Stefan Lucks, and Jakob Wenzel. Memory-demanding password scrambling. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 289–305. Springer, Heidelberg, December 2014. doi:10.1007/978-3-662-45608-8_16.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013. doi:10.1109/FOCS.2013.13.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986. doi:10.1145/6490.6503.

[GHS20] Venkatesan Guruswami, Bernhard Haeupler, and Amirbehshad Shahrasbi. Optimally resilient codes for list-decoding from insertions and deletions. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *52nd ACM STOC*, pages 524–537. ACM Press, June 2020. doi:10.1145/3357713.3384262.

[GL18] Venkatesan Guruswami and Ray Li. Coding against deletions in oblivious and online models. In Artur Czumaj, editor, *29th SODA*, pages 625–643. ACM-SIAM, January 2018. doi:10.1137/1.9781611975031.41.

[GL19] V. Guruswami and R. Li. Polynomial time decodable codes for the binary deletion channel. *IEEE Transactions on Information Theory*, 65(4):2171–2178, 2019. doi:10.1109/TIT.2018.2876861.

[GMPY11] Juan A. Garay, Philip D. MacKenzie, Manoj Prabhakaran, and Ke Yang. Resource fairness and composability of cryptographic protocols. *Journal of Cryptology*, 24(4):615–658, October 2011. doi:10.1007/s00145-010-9080-z.

[GS16] Venkatesan Guruswami and Adam Smith. Optimal rate code constructions for computationally simple channels. *J. ACM*, 63(4):35:1–35:37, September 2016. URL: http://doi.acm.org/10.1145/2936015, doi:10.1145/2936015.

[GS18] Sanjam Garg and Akshayaram Srinivasan. A simple construction of iO for turing machines. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 425–454. Springer, Heidelberg, November 2018. doi:10.1007/978-3-030-03810-6_16.

[GW17] V. Guruswami and C. Wang. Deletion codes in the high-noise and high-rate regimes. *IEEE Transactions on Information Theory*, 63(4):1961–1970, 2017. doi:10.1109/TIT.2017.2659765.

[Hae19] Bernhard Haeupler. Optimal document exchange and new codes for insertions and deletions. In David Zuckerman, editor, *60th FOCS*, pages 334–347. IEEE Computer Society Press, November 2019. doi:10.1109/FOCS.2019.00029.

[HILL99] Johan HÅstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, March 1999. doi:10.1137/S0097539793244708.

[HRS19] Bernhard Haeupler, Aviad Rubinstein, and Amirbehshad Shahrasbi. Near-linear time insertion-deletion codes and $(1+\varepsilon)$-approximating edit distance via indexing. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 697–708. ACM Press, June 2019. doi:10.1145/3313276.3316371.

[HS17] Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings: codes for insertions and deletions approaching the singleton bound. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *49th ACM STOC*, pages 33–46. ACM Press, June 2017. doi:10.1145/3055399.3055498.

[HS18] Bernhard Haeupler and Amirbehshad Shahrasbi. Synchronization strings: explicit constructions, local decoding, and applications. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *50th ACM STOC*, pages 841–854. ACM Press, June 2018. doi:10.1145/3188745.3188940.

[HSS18] Bernhard Haeupler, Amirbehshad Shahrasbi, and Madhu Sudan. Synchronization strings: List decoding for insertions and deletions. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018*, volume 107 of *LIPIcs*, pages 76:1–76:14. Schloss Dagstuhl, July 2018. doi:10.4230/LIPIcs.ICALP.2018.76.

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *ITCS 2015*, pages 163–172. ACM, January 2015. doi:10.1145/2688073.2688105.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st FOCS*, pages 294–304. IEEE Computer Society Press, November 2000. doi:10.1109/SFCS.2000.892118.

[JJ99] Markus Jakobsson and Ari Juels. *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, pages 258–272. Springer US, Boston, MA, 1999. doi:10.1007/978-0-387-35568-9_18.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. *Indistinguishability Obfuscation from Well-Founded Assumptions*, page 60–73. Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3406325.3451093.

[KLM04] Marcos Kiwi, Martin Loebl, and Jiří Matoušek. Expected length of the longest common subsequence for large alphabets. In Martín Farach-Colton, editor, *LATIN 2004: Theoretical Informatics*, pages 302–311, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 419–428. ACM Press, June 2015. doi:10.1145/2746539.2746614.

[KMRS17] Swastik Kopparty, Or Meir, Noga Ron-Zewi, and Shubhangi Saraf. High-rate locally correctable and locally testable codes with sub-polynomial query complexity. *J. ACM*, 64(2):11:1–11:42, 2017.

[KMRZS17] Swastik Kopparty, Or Meir, Noga Ron-Zewi, and Shubhangi Saraf. High-rate locally correctable and locally testable codes with sub-polynomial query complexity. *J. ACM*, 64(2), May 2017. doi:10.1145/3051093.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013. doi:10.1145/2508859.2516668.

[KS16] Swastik Kopparty and Shubhangi Saraf. Guest column: Local testing and decoding of high-rate error-correcting codes. *SIGACT News*, 47(3):46–66, August 2016. doi:10.1145/2993749.2993761.

[KT00] Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *32nd ACM STOC*, pages 80–86. ACM Press, May 2000. doi:10.1145/335305.335315.

[KW03] Iordanis Kerenidis and Ronald de Wolf. Exponential lower bound for 2-query locally decodable codes via a quantum argument. In *35th ACM STOC*, pages 106–115. ACM Press, June 2003. doi:10.1145/780542.780560.

[Lev66] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[Lip94] Richard J. Lipton. A new approach to information theory. In *STACS 94*, pages 699–708, Berlin, Heidelberg, 1994.

[LM18] Huijia Lin and Christian Matt. Pseudo flawed-smudging generators and their application to indistinguishability obfuscation. Cryptology ePrint Archive, Report 2018/646, 2018. https://eprint.iacr.org/2018/646.

[LPST16] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 96–124. Springer, Heidelberg, January 2016. doi:10.1007/978-3-662-49096-9_5.

[LTX20] Shu Liu, Ivan Tjuawinata, and Chaoping Xing. On list decoding of insertion and deletion errors, 2020. arXiv:1906.09705.

[May] Timothy C. May. Timed-release crypto. URL: http://www.hks.net/cpunks/cpunks-0/1460.html.

[MMV11] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 39–50. Springer, Heidelberg, August 2011. doi:10.1007/978-3-642-22792-9_3.

[MPSW05] Silvio Micali, Chris Peikert, Madhu Sudan, and David A. Wilson. Optimal error correction against computationally bounded noise. In Joe Kilian, editor, *Theory of Cryptography*, pages 1–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[MT19] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 620–649. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26948-7_22.

[Nak] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: https://bitcoin.org/bitcoin.pdf.

[OPC15] Rafail Ostrovsky and Anat Paskin-Cherniavsky. Locally decodable codes for edit distance. In Anja Lehmann and Stefan Wolf, editors, *Information Theoretic Security*, pages 236–249, Cham, 2015. Springer International Publishing.

[OPS07] Rafail Ostrovsky, Omkant Pandey, and Amit Sahai. Private locally decodable codes, 2007.

[OPWW15] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 121–145. Springer, Heidelberg, November / December 2015. doi:10.1007/978-3-662-48797-6_6.

[per] Personal communication.

[Per09] C. Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009*, 2009.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, April 1979. doi:10.1145/322123.322138.

[RSW96] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, USA, 1996.

[SB19] Jin Sima and Jehoshua Bruck. Optimal k-deletion correcting codes. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 847–851, 2019. doi:10.1109/ISIT.2019.8849750.

[SS16] Ronen Shaltiel and Jad Silbak. Explicit list-decodable codes with optimal rate for computationally bounded channels. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, pages 45:1–45:38, 2016.

[STV99] M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the xor lemma. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference) (Cat.No.99CB36317)*, pages 4–, 1999. doi:10.1109/CCC.1999.766253.

[Yek08] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1), February 2008. doi:10.1145/1326554.1326555.

[Yek12] Sergey Yekhanin. Locally decodable codes. *Foundations and Trends® in Theoretical Computer Science*, 6(3):139–255, 2012. URL: http://dx.doi.org/10.1561/0400000030, doi:10.1561/0400000030.

A Formal Definitions of Cryptographic Primitives

A.1 Puncturable Pseudorandom Function (PPRF)

Puncturable Pseudorandom Functions (PPRFs) [BW13, KPTZ13, BGI14] are a special class of pseudorandom functions which have proven to be very useful in combination with indistinguishabilty obfuscation. A PPRF consists of three PPT algorithms $F.\mathsf{KeyGen}$, $F.\mathsf{eval}$ and $F.\mathsf{puncture}$.

- $F.\mathsf{KeyGen}(1^\lambda)$ is a randomized algorithm which takes as input the security parameter λ (in unary) and outputs a PRF secret key $K \in \mathcal{K}$.
- $F.\mathsf{puncture}(K, x_1, \dots, x_k)$ is a randomized algorithm which takes as input the PRF secret key K and a list of inputs $x_1, \dots, x_k \in \mathcal{X}$, and outputs a punctured key $K\{x_1, \dots, x_k\} \in \mathcal{K}_p$.
- $F.\mathsf{eval}(K, x')$: is a randomized algorithm which takes as input a PRF key $K \in \mathcal{K} \cup \mathcal{K}_p$ and outputs a pseudorandom string $y \in \mathcal{Y} \cup \{\perp\}$.

For correctness we require that $F.\mathsf{eval}(K, x) = F.\mathsf{eval}(K', x)$ whenever $K \leftarrow F.\mathsf{KeyGen}(1^\lambda)$, $K' \leftarrow F.\mathsf{puncture}(K, x_1, \dots, x_k)$ and $x \notin \{x_1, \dots, x_k\}$ and we have $F.\mathsf{eval}(K', x) = \perp$ whenever $x \in \{x_1, \dots, x_k\}$ and $K' \leftarrow F.\mathsf{puncture}(K, x_1, \dots, x_k)$. For simplicity we use the notation $F(K, x) := F.\mathsf{eval}(K, x)$ and we use the notation $K\{x_1, \dots, x_k\}$ to denote the punctured key $F.\mathsf{puncture}(K, x_1, \dots, x_k)$. Intuitively, the punctured key $K\{x_1, \dots, x_k\}$ allows one to evaluate the PRF everywhere on all inputs excluding $x_1, \dots, x_k$.

For security we require that an attacker who has the punctured key $K\{x_1, \dots, x_k\}$ cannot infer the value $F.\mathsf{eval}(K, x)$ for $x \in \{x_1, \dots, x_k\}$; i.e., any PPT attacker given $K\{x_1, \dots, x_k\}$ cannot distinguish $F.\mathsf{eval}(K, x_1), \dots, F.\mathsf{eval}(K, x_k)$ from random strings $y_1, \dots, y_k \xleftarrow{\$} \mathcal{Y}$. Formally, security is defined based on the experiment $\mathsf{SS} - \mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, x)$ described in Figure 4 which is a game between the honest challenger $\mathcal{C}$ and a PPT adversary $\mathcal{A}$.

Definition A.1 (Selectively secure puncturable PRF). *The function F is selectively secure puncturable PRF if for all constants $k > 0$, all inputs $\mathbf{x} = (x_1, \dots, x_k) \in \mathcal{X}$, all PPT adversary $\mathcal{A}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that the attacker's advantage is at most:*

$$\mathsf{Adv}^{\mathsf{ss-pprf}}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) \leqslant \mathsf{negl}(\lambda)$$

For concrete security we say that PPRF is $(t(\cdot), \varepsilon(\cdot))$-secure PPRF if $\mathsf{Adv}^{\mathsf{ss-pprf}}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) \leqslant \varepsilon(\lambda)$ for any security parameter $\lambda > 0$, $k > 0$, $\mathbf{x} = (x_1, \dots, x_k) \in \mathcal{X}$ and any adversary $\mathcal{A}$ running it time at most $t(\lambda)$.

A.2 Indistinguishable Obfuscation

We give the formal definition of indistinguishability obfuscation.

Definition A.2 (Indistinguishable Obfuscation). *We say that a PPT algorithm $i\mathcal{O}$ is an* indistinguishability obfuscator *for a circuit class $\mathcal{C}_\lambda$ if the following conditions are satisfied*

Correctness. *For all $\lambda \in \mathbb{N}$ as the security parameter, all the input values x and the polynomial circuit family $\mathcal{C}_\lambda$ we have $\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(C)] = 1$.*

Indistinguishability. *For any PPT distinguisher, say $\mathcal{D}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all security parameters $\lambda > 0$ and all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$:*

$$\mathsf{Adv}_{\mathcal{D}}(\lambda) := \Big| \Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_0)) = 1] - \Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_1)) = 1] \Big| < \mathsf{negl}(\lambda). \tag{3}$$

The Selectively secure PPRF experiment $\mathsf{SS}-\mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, \mathbf{x})$

- **Init**.
  1. The challenger $\mathcal{C}$ runs generates a PRF key $K \leftarrow F.\mathsf{KeyGen}(1^\lambda)$ and sets $K\{\mathbf{x}\} \leftarrow F.\mathsf{puncture}(K, \mathbf{x})$.
  2. $\mathcal{C}$ samples $b \in_R \{0,1\}$.
  3. For each $i \in [k]$ the challenger $\mathcal{C}$ sets $y_i := F(K, x)$ if $b = 0$; otherwise $y_i \xleftarrow{\$} \mathcal{Y}$.
  4. $\mathcal{C}$ sends $K\{\mathbf{x}\}$ and $y_1, \ldots y_k$ to the adversary $\mathcal{A}$.
- **Guess**. The adversary $\mathcal{A}$ is given $K\{\mathbf{x}\}, \mathbf{x}, y$ as input and outputs a guess b' for b. If $b = b'$ the experiment outputs $\mathsf{SS}-\mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) = 1$ indicating that $\mathcal{A}$ wins the game; otherwise the experiment outputs $\mathsf{SS}-\mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) = 0$.

  We define the advantage of $\mathcal{A}$ in this experiment as follows:

$$\mathsf{Adv}^{\mathsf{ss-pprf}}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) = |\Pr[\mathsf{SS}-\mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, \mathbf{x}) = 1] - 1/2|.$$

Figure 4: Details description of the experiment $\mathsf{SS}-\mathsf{PPRF}_{\mathcal{A},F}(1^\lambda, x)$.

For concrete security we say that $i\mathcal{O}$ is $(t(\cdot), \varepsilon(\cdot))$-secure if $\mathsf{Adv}_{\mathcal{D}}(\lambda) \leqslant \varepsilon(\lambda)$ for any security parameter $\lambda > 0$ and any distinguisher $\mathcal{D}$ running it time at most $t(\lambda)$.

B Proof of Security for Theorem 4.8

We prove the security of Theorem 4.8. We restate the theorem here as a reminder.

Theorem 4.8. *Let t be a polynomial and let g be a function. Let $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ be a succinct randomized encoding scheme. If there exists a g'-strong memory-hard language $\mathcal{L} \in \mathsf{SC}_t$ for*

$$g'(t(\lambda), \lambda) := g(t(\lambda), \lambda) + 2 \cdot p_{\mathsf{sRE}}\big(\log(t(\lambda)), \lambda\big)^2 + 2 \cdot p_{\mathsf{SC}}\big(\log(t(\lambda)), \log(\lambda)\big)^2 + O(\lambda),$$

then Construction 4.7 is a g-memory hard puzzle. Here, p_{sRE} and p_{SC} are fixed polynomials for the run-times of $\mathsf{sRE.Enc}$ and the uniform machine constructing the uniform succinct circuit of $\mathcal{L}$, respectively.

Suppose that Construction 4.7 is not a g-memory hard puzzle. Then for every polynomial t' there exists a polynomial $t > t'$ and a PRAM algorithm A with $\mathsf{aAT}(A, \lambda) < g(t(\lambda), \lambda)$, for every negligible function μ there exists $\lambda \in \mathbb{N}$ and $s_0, s_1 \in \{0,1\}^\lambda$ such that

$$\Pr[A(Z_b, Z_{1-b}, s_0, s_1) = b] > \frac{1}{2} + \mu(\lambda),$$

where the probability is taken over $b \xleftarrow{\$} \{0,1\}$ and $Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ for $i \in \{0,1\}$. We construct a PRAM adversary B that breaks the memory-hardness of some g'-strong memory-hard language $\mathcal{L} \in \mathsf{SC}_t$, for $t := t(\lambda)$.

Fix $t', t, A, \varepsilon, \lambda, s_0$, and s_1, where $\varepsilon(\lambda)$ is the advantage of A. Note that $\varepsilon(\lambda) = 1/\operatorname{poly}(\lambda)$. We specify sub-routines that the adversary B will use.

1. Let $\mathcal{L} \in \mathsf{SC}_t$ be a g'-strong memory-hard language. By assumption there exists a PRAM algorithm $\mathcal{A}_{\mathcal{L}}$ such that on input λ and t, $\mathcal{A}_{\mathcal{L}}(t, \lambda)$ outputs succinct circuit $C^{\mathsf{sc}}_{t,\lambda}$ in time $O(|C^{\mathsf{sc}}_{t,\lambda}|)$ such that circuit $C_{t,\lambda} = \mathsf{FullCirc}(C^{\mathsf{sc}}_{t,\lambda})$ decides $\mathcal{L}_\lambda$. By assumption, $|C_{t,\lambda}| = t \cdot \text{poly}(\lambda, \log(t))$ and $|C^{\mathsf{sc}}_{t,\lambda}| = \text{polylog}(|C_{t,\lambda}|) = \text{polylog}(\lambda, t)$. Let p_{SC} denote the polynomial such that $\mathcal{A}_{\mathcal{L}}(t, \lambda)$ runs in time $p_{\mathsf{SC}}(\log(\lambda), \log(t))$. Note that $\mathsf{aAT}(\mathcal{A}_{\mathcal{L}}, \lambda) \leqslant p_{\mathsf{SC}}(\log(\lambda), \log(t))^2$.
2. For $a, b \in \{0,1\}^\lambda$, define a circuit $\widetilde{C}_{a,b}$ such that for every $x \in \{0,1\}^\lambda$

$$\widetilde{C}_{a,b}(x) = \begin{cases} a & C_{t,\lambda}(x) = 1 \\ b & C_{t,\lambda}(x) = 0 \end{cases},$$

where $C_{t,\lambda}$ decides the language $\mathcal{L}_\lambda$. Note that since $C_{t,\lambda}$ is uniformly succinct, the circuit $\widetilde{C}_{a,b}$ is also uniformly succinct. Thus there exists a PRAM algorithm $\widetilde{\mathcal{A}}$ such that on input t, λ, a, b constructs circuit $\widetilde{C}^{\mathsf{sc}}_{a,b}$ such that $\widetilde{C}_{a,b} = \mathsf{FullCirc}(\widetilde{C}^{\mathsf{sc}}_{a,b})$. Further, $\mathsf{aAT}(\widetilde{\mathcal{A}}, \lambda) \leqslant O(\lambda) + p_{\mathsf{SC}}(\log(\lambda), \log(t))^2$.

We now define PRAM adversary B to break the memory-hard language assumption for language $\mathcal{L}$.

PRAM algorithm B

Input: $x \in \{0,1\}^\lambda$.
Hardcoded: $s_0, s_1 \in \{0,1\}^\lambda$, t, λ, PRAM algorithms A and $\widetilde{\mathcal{A}}$, and $\mathsf{sRE.Enc}$.

1. Obtain succinct circuits $\widetilde{C}^{\mathsf{sc}}_i := \widetilde{C}^{\mathsf{sc}}_{s_i, s_{1-i}} = \widetilde{\mathcal{A}}(t, \lambda, s_i, s_{1-i})$ for $i \in \{0,1\}$.
2. Obtain $\widetilde{Z}_i \leftarrow \mathsf{sRE.Enc}(1^\lambda, \widetilde{C}^{\mathsf{sc}}_i, x, G)$ for $i \in \{0,1\}$ where $G = |\mathsf{FullCirc}(\widetilde{C}^{\mathsf{sc}}_i)| = t \cdot \text{poly}(\lambda, \log(t))$.
3. Sample $b \xleftarrow{\$} \{0,1\}$.
4. Obtain $b' \leftarrow A(\widetilde{Z}_b, \widetilde{Z}_{1-b}, s_0, s_1)$.
5. Output $b' = b$.

Figure 5: PRAM adversary B for breaking memory-hard language $\mathcal{L}$.

We argue that B decides the language $\mathcal{L}$ with non-negligible advantage. We analyze the probability that $B(x) = 1$ for $x \in \mathcal{L}_\lambda$ and note that the case for $x \notin \mathcal{L}_\lambda$ is symmetric. By construction, we have that $B(x) \leftarrow (b = A(\widetilde{Z}_b, \widetilde{Z}_{1-b}, s_0, s_1))$ for $b \xleftarrow{\$} \{0,1\}$ and $\widetilde{Z}_i \leftarrow \mathsf{sRE.Enc}(1^\lambda, \widetilde{C}^{\mathsf{sc}}_i, x, G)$. By construction, the algorithm $\mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ constructs machine M_{t,s_i} such that on any input $x' \in \{0,1\}^\lambda$, $M_{t,s_i}(x')$ delays for t steps then outputs s_i. Then $\mathsf{Puz.Gen}$ outputs $Z_i \leftarrow \mathsf{sRE.Enc}(1^\lambda C^{\mathsf{sc}}_{t,s_i}, G_M)$ where $G_M = |\mathsf{FullCirc}(C^{\mathsf{sc}}_{t,s_i})|$. Note that M_{t,s_i} runs in time t and space $O(\lambda + \log(t))$. By Lemma 3.6 this implies that $G_M = \widetilde{O}(t \cdot (\lambda + \log(t))) = t \cdot \text{poly}(\lambda, \log(t))$ and that $|C^{\mathsf{sc}}_{t,s_i}| = O(\lambda + \log(t))$.

By the security of sRE, there exists a PPT simulator $\mathcal{S}$ such that for any poly-sized adversary $\mathcal{A}_{\mathsf{sRE}}$ there exists a negligible function ϑ such that for all $\lambda \in \mathbb{N}$, succinct circuits C^{sc}, input x, and $G = |\mathsf{FullCirc}(C^{\mathsf{sc}})|$ we have

$$\left| \Pr\left[\mathcal{A}_{\mathsf{sRE}}(\widehat{C}^{\mathsf{sc}}_{x',t}) = 1\right] - \Pr\left[\mathcal{A}_{\mathsf{sRE}}(\mathcal{S}(1^\lambda, y', C^{\mathsf{sc}}, G)) = 1\right]\right| \leqslant \vartheta(\lambda),$$

where $\widehat{C}^{\mathsf{sc}}_{x',t} \leftarrow \mathsf{sRE.Enc}(1^\lambda, C^{\mathsf{sc}}, x', t)$ and y' is the output of $\mathsf{FullCirc}(C^{\mathsf{sc}})(x)$. Note that by construction of the memory-hard puzzle, the adversary A is also an adversary against the succinct randomized encoding scheme. This implies that for $b \xleftarrow{\$} \{0,1\}$ we have

$$\Pr\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \;:\; Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_i)\right] =$$

$$\Pr\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \ : \ Z_i \leftarrow \mathsf{sRE.Enc}(1^\lambda, C^{\mathsf{sc}}_{t,\lambda}, 0^\lambda, t)\right] =$$
$$\Pr\left[A(\widehat{S}_b, \widehat{S}_{1-b}, s_0, s_1) = b \ : \ \widehat{S}_i \leftarrow \mathcal{S}(1^\lambda, s_i, C^{\mathsf{sc}}_{t,\lambda}, G_M)\right] \pm \vartheta(\lambda), \tag{4}$$

and

$$\Pr[B(x) = 1] =$$
$$\Pr\left[A(\widetilde{Z}_b, \widetilde{Z}_{1-b}, s_0, s_1) = b \ : \ \widetilde{Z}_i \leftarrow \mathsf{sRE.Enc}(1^\lambda, \widetilde{C}^{\mathsf{sc}}_i, x, G)\right] =$$
$$\Pr\left[A(\widetilde{S}_b, \widetilde{S}_{1-b}, s_0, s_1) = b \ : \ \widetilde{S}_i \leftarrow \mathcal{S}(1^\lambda, s_i, \widetilde{C}^{\mathsf{sc}}_i, G)\right] \pm \vartheta(\lambda). \tag{5}$$

Since G_M and G are both of asymptotic size $t \cdot \mathrm{poly}(\lambda, \log(t))$, we have that Eqs. (4) and (5) are distinguishable by A with advantage at most $\pm\vartheta(\lambda)$. By assumption, A correctly outputs b with advantage at least $\varepsilon(\lambda)$. Observe that

$$(\widetilde{Z}_b, \widetilde{Z}_{1-b}, s_0, s_1) \equiv (Z_b, Z_{1-b}, s_0, s_1) \qquad x \in \mathcal{L};$$
$$(\widetilde{Z}_b, \widetilde{Z}_{1-b}, s_0, s_1) \equiv (Z_{1-b}, Z_b, s_0, s_1) \qquad x \notin \mathcal{L},$$

where the above distributions are identical over $b \xleftarrow{\$} \{0,1\}$ and the random coins of $\mathsf{sRE.Enc}$ since $\mathsf{sRE.Dec}(\widetilde{Z}_i) = \mathsf{Puz.Sol}(Z_i)$ for $x \in \mathcal{L}$ and $\mathsf{sRE.Dec}(\widetilde{Z}_i) = \mathsf{Puz.Sol}(Z_{1-i})$ for $x \notin \mathcal{L}$. This implies for $x \in \mathcal{L}$

$$\Pr[B(x) = 1] \geqslant \Pr_{b \xleftarrow{\$} \{0,1\}}\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \ : \ Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_i))\right] - 2 \cdot \vartheta(\lambda)$$
$$> \frac{1}{2} + \varepsilon(\lambda) - 2 \cdot \vartheta(\lambda),$$

and for $x \notin \mathcal{L}$

$$\Pr[B(x) = 0] \geqslant \Pr_{b \xleftarrow{\$} \{0,1\}}\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \ : \ Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_i))\right] - 2 \cdot \vartheta(\lambda)$$
$$> \frac{1}{2} + \varepsilon(\lambda) - 2 \cdot \vartheta(\lambda),$$

This implies that B decides $\mathcal{L}$ with advantage $\delta(\lambda) = \varepsilon(\lambda) - 2 \cdot \vartheta(\lambda)$. Since $\varepsilon(\lambda) = 1/\mathrm{poly}(\lambda)$, we have that $\delta(\lambda)$ is a non-negligible function.

Finally, to break the g'-strong memory-hard language assumption, we show that $\mathsf{aAT}(B, \lambda) < g'(t, \lambda)$. First note that $\mathsf{sRE.Enc}(1^\lambda, \widetilde{C}^{\mathsf{sc}}_i, x, G)$ runs in time $\mathrm{poly}(|\widetilde{C}^{\mathsf{sc}}_i|, \lambda, \log(G))$. Then since $|\widetilde{C}^{\mathsf{sc}}_i| = \mathrm{polylog}(\lambda, t)$ and $G = t \cdot \mathrm{poly}(\lambda, \log(t))$, we have that the runtime of $\mathsf{sRE.Enc}$ is $\mathrm{poly}(\lambda, \log(t))$. By assumption we have that $\mathsf{sRE.Enc}$ runs in time $p_{\mathsf{sRE}}(\lambda, \log(t))$. Now by construction of B we have that

$$\begin{aligned}\mathsf{aAT}(B, \lambda) &< 2 \cdot \mathsf{aAT}(\widetilde{A}, \lambda) + 2 \cdot p_{\mathsf{sRE}}(\lambda, \log(t))^2 + g'(t, \lambda) \\ &\leqslant O(\lambda) + 2 \cdot p_{\mathsf{SC}}(\log(\lambda), \log(t))^2 + \cdot p_{\mathsf{sRE}}(\lambda, \log(t))^2 + g(t(\lambda), \lambda) \\ &= g'(t, \lambda).\end{aligned}$$

This implies that B breaks the g-strong memory-hard language assumption, completing the proof.

C Proof of Security for Theorem 4.9

We prove the security of Theorem 4.9. We restate the theorem here as a reminder.

Theorem 4.9. *Let t be a polynomial and let g be a function. Let* $\mathsf{sRE} = (\mathsf{sRE.Enc}, \mathsf{sRE.Dec})$ *be a $(g, s, \varepsilon_{\mathsf{sRE}})$-secure succinct randomized encoding scheme for $g := g(t(\lambda), \lambda)$ and $s(\lambda) := t(\lambda) \cdot \mathrm{poly}(\lambda, \log(t(\lambda)))$ such that p_{sRE} is a fixed polynomial for the runtime of* $\mathsf{sRE.Enc}$. *Let $\varepsilon(\lambda) = 1/\mathrm{poly}(\lambda) > 3\varepsilon_{\mathsf{sRE}}(\lambda)$ be fixed.*

If there exists a $(g', \varepsilon_{\mathcal{L}})$-weakly memory-hard language $\mathcal{L} \in \mathsf{SC}_t$ for

$$g'(t(\lambda), \lambda) := \left[g(t(\lambda), \lambda) + 2 \cdot p_{\mathsf{sRE}}\big(\log(t(\lambda)), \lambda\big)^2 + 2 \cdot p_{\mathsf{SC}}\big(\log(t(\lambda)), \log(\lambda)\big)^2 + O(\lambda)\right] \cdot \Theta(1/\varepsilon(\lambda)),$$

and some constant $\varepsilon_{\mathcal{L}} \in (0, 1/2)$, then Construction 4.7 is a (g, ε)-weakly memory-hard puzzle. Here, p_{SC} is a fixed polynomial for the runtime of the uniform machine constructing the uniform succinct circuit for $\mathcal{L}$.

Suppose that Construction 4.7 is not (g, ε)-memory hard. Then for any polynomial t' there exists polynomial $t > t'$ and a PRAM algorithm A with $\mathsf{aAT}(A, \lambda) < g(t(\lambda), \lambda)$, there exists λ_0 such that for all $\lambda > \lambda_0$ there exists $s_0, s_1 \in \{0,1\}^\lambda$ such that

$$\Pr\left[A(Z_b, Z_{1-b}, s_0, s_1) = b\right] > \frac{1}{2} + \varepsilon(\lambda),$$

where the probability is taken over $b \xleftarrow{\$} \{0,1\}$ and $Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s_i)$ for $i \in \{0,1\}$. We construct a PRAM adversary $\mathcal{B}$ that breaks the memory-hardness of some g'-weakly memory-hard language $\mathcal{L} \in \mathsf{SC}_t$.

Fix $t', t, A, \varepsilon, \lambda_0, \lambda, s_0$, and s_1. The remainder of the proof is nearly identical to the proof presented in Appendix B, however, the analysis is different to account for the concrete security requirements. In particular, we first construct PRAM adversary B exactly as in Figure 5. We then appeal to the concrete security requirement of the succinct randomized encoding. That is, there exists a probabilistic simulator $\mathcal{S}$ and polynomial $p_{\mathcal{S}}$ such that for every λ, every adversary $\mathcal{A}_{\mathsf{sRE}}$ running in time $g(t(\lambda), \lambda)$, every succinct circuit C' such that $|\mathsf{FullCirc}(C')| = G \leqslant s(\lambda)$, and every input $x \in \{0,1\}^\lambda$, we have

$$\left|\Pr\left[\mathcal{A}_{\mathsf{sRE}}(\widehat{C}_{x,G}) = 1\right] - \Pr\left[\mathcal{A}_{\mathsf{sRE}}(\mathcal{S}(1^\lambda, y, C', G)) = 1\right]\right| \leqslant \varepsilon_{\mathsf{sRE}}(\lambda),$$

where $\widehat{C}_{x,G} \leftarrow \mathsf{sRE.Enc}(1^\lambda, C', x, G)$, $y = \mathsf{FullCirc}(C')(x)$, and $\mathcal{S}$ runs in time at most $G \cdot p_{\mathcal{S}}(\lambda)$. We remark that the adversary A against our puzzle is also and adversary against the specified succinct randomized encoding scheme. In particular, adversary A has $\mathsf{aAT}(t, \lambda) < g(t(\lambda), \lambda)$, which upper bounds the running time of A, and the puzzle constructs a succinct randomized encoding of the succinct circuit representing Turing machine M_{t,s_i}. This succinct circuit C^{sc}_{t,s_i} represents larger circuit C_{t,s_i} of size $t \cdot \mathrm{poly}(\lambda, \log(t))$. By the same argument as in Appendix B, we have that for $x \in \mathcal{L}$

$$\begin{aligned}\Pr[B(x) = 1] &\geqslant \Pr_{b \xleftarrow{\$} \{0,1\}}\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \colon Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_i)\right] - 2 \cdot \varepsilon_{\mathsf{sRE}}(\lambda) \\ &> \frac{1}{2} + \varepsilon(\lambda) - 2 \cdot \varepsilon_{\mathsf{sRE}}(\lambda),\end{aligned}$$

and for $x \notin \mathcal{L}$

$$\begin{aligned}\Pr[B(x) = 0] &\geqslant \Pr_{b \xleftarrow{\$} \{0,1\}}\left[A(Z_b, Z_{1-b}, s_0, s_1) = b \colon Z_i \leftarrow \mathsf{Puz.Gen}(1^\lambda, t, s_i)\right] - 2\varepsilon_{\mathsf{sRE}}(\lambda) \\ &> \frac{1}{2} + \varepsilon(\lambda) - 2 \cdot \varepsilon_{\mathsf{sRE}}(\lambda).\end{aligned}$$

Thus B decides $\mathcal{L}$ with advantage $\delta(\lambda) = \varepsilon(\lambda) - 2\varepsilon_{\mathsf{sRE}}(\lambda)$. By the same analysis as in Appendix B, we have that

$$\mathsf{aAT}(B, \lambda) < O(\lambda) + 2 \cdot p_{\mathsf{SC}}(\log(\lambda), \log(t))^2 + 2 \cdot p_{\mathsf{sRE}}(\lambda, \log(t))^2 + g(t(\lambda), \lambda).$$

Finally, we obtain adversary $\mathcal{B}$ which has advantage $1/4$ for deciding $\mathcal{L}$ by amplification. That is, we run adversary B in parallel $\Theta(1/\delta(\lambda))$ times and output the majority answer. Note that the initial $\Theta(1/\delta)$ amplification increases the advantage so some constant that depends on δ, after which we amplify $\Theta(1)$ additional times to reach advantage $1/4$. This increases the aAT complexity by a multiplicative $\Theta(1/\delta(\lambda))$, which implies

$$\begin{aligned}\mathsf{aAT}(\mathcal{B},\lambda) &< \big(O(\lambda) + 2\cdot p_{\mathsf{SC}}(\log(\lambda),\log(t))^2 + 2\cdot p_{\mathsf{sRE}}(\lambda,\log(t))^2 + g(t(\lambda),\lambda)\big)\cdot\Theta(1/\delta(\lambda))\\ &= g'(t(\lambda),\lambda).\end{aligned}$$

Thus $\mathcal{B}$ breaks the g-weakly memory-hard language assumption.

D Proof of Theorem 5.4

Theorem 5.4. *Let t be a polynomial and let g be a function. If there exists a $(t_{\mathrm{PPRF}},\varepsilon_{\mathrm{PPRF}})$-secure PPRF family, a $(t_{i\mathcal{O}},\varepsilon_{i\mathcal{O}})$-secure $i\mathcal{O}$ scheme, and a $(g,\varepsilon_{\mathrm{MHP}})$-memory hard puzzle for $g(t(\lambda),\lambda)\leqslant\min\{t_{\mathrm{PPRF}}(\lambda),t_{i\mathcal{O}}(\lambda)\}$, then Construction 5.3 is a one-time $(g',\varepsilon_{\mathrm{MHF}})$-MHF for*

$$g'(t(\lambda),\lambda) = g(t(\lambda),\lambda)/p(\log(t(\lambda)),\lambda)^2,$$

where $\varepsilon_{\mathrm{MHF}}(\lambda) = 2\cdot\varepsilon_{\mathrm{MHP}}(\lambda)+3\cdot\varepsilon_{\mathrm{PPRF}}(\lambda)+\varepsilon_{i\mathcal{O}}(\lambda)$ and $p(\log(t),\lambda)$ is a fixed polynomial which depends on the efficiency of underlying memory-hard puzzle and $i\mathcal{O}$.

Overview. We use a hybrid argument to prove that Construction 5.3 is secure. We introduce hybrids H_0, H_1, H_2 and H_3 where H_0 is the original construction and we can show that any attacker wins the MHF game in H_3 with negligible probability. Indistinguishability of the hybrids will follow from $i\mathcal{O}$ security, PPRF security, and MHP security, respectively.

In the rest of this section, we first define the relevant hybrids, then we will prove their indistinguishability, and finally prove the security of the proposed scheme.

D.1 Defining hybrids

In what follows, we will define the hybrids H_0, H_1, H_2 and H_3 describing the differences between each pair H_i and H_{i+1}. Hybrid H_0 is the real world where we use Construction 5.3 without modification. We use notation $\mathsf{prog}[K_1,K_2,K_3,\lambda,t](x,s)$ to represent the program prog with hardcoded values K_1,K_2,K_3,λ,t which takes (x,s') as input.

D.1.1 Hybrid H_0

Our first hybrid H_0 (real) uses the original construction Construction 5.3 without modification i.e., we set $\mathsf{pp}_{H_0}\leftarrow \mathrm{MHF}.\mathsf{Setup}(1^\lambda)$. For convenience we remind the reader of Construction 5.3 in Figure 6 below.

D.1.2 Hybrid H_1

This hybrid is similar to H_0 except that we modify MHF.Setup to puncture the keys K_1,K_2,K_3 at x_0 and x_1, hardcode the puzzles Z_0,Z_1 (resp. solutions s_0,s_1 and outputs h_0,h_1) corresponding to x_0 and x_1. Specifically we hardcode the values $s_i = F(K_1,x_i)$, $h_i = F(K_3,x_i)$ and $Z_i := \mathsf{Puz.Gen}(1^\lambda,s_i;r_i)$ for $i\in\{0,1\}$ where $r_i := F(K_2,x)$. We also modify prog to and equivalent program prog_1 that uses the puctured keys $K_i\{x_0,x_1\}$ along with the hardcoded values Z_0,Z_1. MHF.Setup is defined below

$\underline{\mathsf{pp}\leftarrow \mathrm{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))}$

$\underline{\mathsf{pp} \leftarrow \text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))}$	$\underline{\mathsf{prog}[K_1, K_2, K_3, \lambda, t](x, s')}$
1. `Sample keys` $K_i \xleftarrow{\$} \{0,1\}^\lambda$ `for` $i \in [3]$	**Internal (hardcoded) state**: the set of secret PRF keys K_1, K_2, K_3, and hardness parameter λ and $t = t(\lambda)$.
2. `Output` $\mathsf{pp} := i\mathcal{O}(\mathsf{prog}[K_1, K_2, K_3, \lambda, t(\lambda)])$	1. `Compute` $s := F(K_1, x)$ `and` $r := F(K_2, x)$
$\underline{h = \text{MHF}.\mathsf{Eval}(\mathsf{pp}, x)}$	2. `if` $s' = \varnothing$,
1. `Compute` $Z \leftarrow \mathsf{pp}(x, \varnothing)$ // $Z = \mathsf{Puz.Gen}(1^\lambda, t, F(K_1, x); F(K_2, x))$	- `return` $Z := \mathsf{Puz.Gen}(1^\lambda, t, s; r)$
2. `Compute` $r' \leftarrow \mathsf{Puz.Sol}(Z)$	3. `else if` $s = s'$, `return` $h = F(K_3, x)$
3. `Compute` $h \leftarrow \mathsf{pp}(x, r')$ // $h = F(K_3, x)$	4. `else return` $\perp$
4. `return` h	

Figure 6: Reminder of Construction 5.3: MHF.Setup, MHF.Eval, and prog.

1. Sample secret keys $K_i \xleftarrow{\$} \{0,1\}^\lambda$ for $i \in [3]$.
2. Generate punctured keys $K_i\{x_0, x_1\} \leftarrow F.\mathsf{Puncture}(K_i, x_0, x_1)$ for each $i \in [3]$.
3. Compute hardcoded values $s_i = F.\mathsf{Eval}(K_1, x_i)$,$r_i := F(K_2, x)$, $Z_i := \mathsf{Puz.Gen}(1^\lambda, s_i; r_i)$ and $h_i = F(K_3, x_i)$ for $i \in \{0, 1\}$.
4. Output $\mathsf{pp} := i\mathcal{O}(\mathsf{prog}_1[K_1\{x_1, x_2\}, K_2\{x_1, x_2\}, K_3\{x_1, x_2\}, s_0, s_1, h_0, h_1, Z_0, Z_1, \lambda, t = t(\lambda)])$.

We replace the original program prog with the program $\mathsf{prog}_1[K_{j\in[3]}\{x_0, x_1\}, s_0, s_1, h_0, h_1, Z_0, Z_1]$ described in Figure 7 and then set $\mathsf{pp}_{H_1} = i\mathcal{O}(\mathsf{prog}_1)$. Here, we stress that the hardcoded values are selected to ensure that prog and prog_1 are functionally equivalent i.e., $Z_0 := \mathsf{Puz.Gen}(1^\lambda, s_0; r_0)$, $Z_1 := \mathsf{Puz.Gen}(1^\lambda, s_1; r_1)$, $s_i = F.\mathsf{Eval}(K_1, x_i)$, $r_i = F.\mathsf{Eval}(K_2, x_i)$ and $h_i = F.\mathsf{Eval}(K_3, x_i)$ for $i \in \{0, 1\}$. Intuitively, indistinguishability of hybrids 1 and 2 follows from $i\mathcal{O}$ security.

The key difference between prog_1 and prog (highlighted in blue) is that the PPRF keys K_1, K_2 and K_3 are replaced with the punctured keys $K_1\{x_0, x_1\}$, $K_2\{x_0, x_1\}$, and $K_3\{x_0, x_1\}$ respectively. The missing values are hard coded so that prog_1 can still mimic prog exactly even when the input is x_0 or x_1. By appealing to $i\mathcal{O}$ security we can argue that any attacker running in time at most $t_{i\mathcal{O}}(\lambda)$ can distinguish H_0 and H_1 with probability at most $\varepsilon_{i\mathcal{O}}(\lambda)$.

D.1.3 Hybrid H_2

The key difference between hybrid 2 and hybrid 1 is that we now select the hardcoded values s_0, s_1, h_0, h_1, Z_0, and Z_1 randomly — independent of the PRF keys K_1, K_2, K_3. In particular, for $i \in \{0, 1\}$ we sample s_i, h_i, r_i *uniformly* at random and then set $Z_i = \mathsf{Puz.Gen}(1^\lambda, s_i; r_i)$. We then set

$$\mathsf{pp}_{H_2} \leftarrow i\mathcal{O}(\mathsf{prog}_1[K_1\{x_0, x_1\}, K_3\{x_0, x_1\}, K_3\{x_0, x_1\}, s_0, s_1, h_0, h_1, Z_0, Z_1, \lambda, t(\lambda)]) .$$

The modified program MHF.Setup is defined below

$\underline{\mathsf{pp} \leftarrow \text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))}$

1. Sample secret keys $K_i \xleftarrow{\$} \{0,1\}^\lambda$ for $i \in [3]$.

$\mathsf{prog}_1[K_{j\in[3]}\{x_0,x_1\}, s_0, s_1, h_0, h_1, Z_0, Z_1, \lambda, t(\lambda)](x, s')$

Internal (hardcoded) state: punctured PRF keys $K_1\{x_0,x_1\}$, $K_2\{x_0,x_1\}$, $K_3\{x_0,x_1\}$, h_0, h_1, s_0, s_1, Z_0, Z_1, hardness parameters λ, t

Input: x, s'.

```
1. if x ∈ {x_0, x_1}
       if s' = ∅
           if x = x_0, return Z_0, else, return Z_1
       else if x = x_0 and s' = s_0, return h_0
       else if x = x_1 and s' = s_1, return h_1
       else return ⊥

2. s := F(K_1{x_0, x_1}, x), r := F(K_2{x_0, x_1}, x)

3. if s' = ∅
       return Z := Puz.Gen(g(t(λ)), s; r)

4. if s = s'
       return h = F(K_3{x_0, x_1}, x)

5. return ⊥
```

Figure 7: Description of the program $\mathsf{prog}_1[K_{j\in[3]}\{x_0,x_1\}, s_0, s_1, h_0, h_1, Z_0, Z_1]$.

2. Generate punctured keys $K_i\{x_0,x_1\} \leftarrow F.\mathsf{Puncture}(K_i, x_0, x_1)$ for each $i \in [3]$.
3. Sample s_i, h_i, r_i randomly and compute $Z_i := \mathsf{Puz.Gen}(1^\lambda, s_i; r_i)$ for $i \in \{0,1\}$.
4. Output $\mathsf{pp} := i\mathcal{O}(\mathsf{prog}_1[K_1\{x_1,x_2\}, K_2\{x_1,x_2\}, K_3\{x_1,x_2\}, s_0, s_1, h_0, h_1, Z_0, Z_1, \lambda, t = t(\lambda)])$.

Intuitively, indistinguishability follows from puncturable PRF security. In particular, any attacker running in time at most $t_{PPRF}(\lambda)$ distinguishes H_1 and H_2 with advantage at most $3\varepsilon_{PPRF}(\lambda)$ since we punctured three PRF keys K_1, K_2, K_3.

D.1.4 Hybrid H_3

In this hybrid the values $s_0, s_1, h_0, h_1, Z_0, Z_1$ are selected exactly as in hybrid 2. We then flip a random coin $b' \in \{0,1\}$ and set $\mathsf{pp}_{H_3} \leftarrow i\mathcal{O}(\mathsf{prog}_1[K_1\{x_0,x_1\}, K_3\{x_0,x_1\}, K_3\{x_0,x_1\}, s_0, s_1, h_0, h_1, Z_{b'}, Z_{1-b'}, \lambda, t(\lambda)])$. If $b' = 0$ then we follow hybrid 2 exactly, but if $b' = 1$ the puzzles Z_0 and Z_1 are swapped. The modified program MHF.Setup is defined below

$\underline{\mathsf{pp} \leftarrow \text{MHF}.\mathsf{Setup}(1^\lambda, t(\lambda))}$

1. Sample secret keys $K_i \xleftarrow{\$} \{0,1\}^\lambda$ for $i \in [3]$.
2. Generate punctured keys $K_i\{x_0,x_1\} \leftarrow F.\mathsf{Puncture}(K_i, x_0, x_1)$ for each $i \in [3]$.

3. Sample s_i, h_i, r_i randomly and compute $Z_i := \mathsf{Puz.Gen}(1^\lambda, s_i; r_i)$ for $i \in \{0,1\}$.

4. Sample a random bit $b' \in \{0,1\}$.

5. Output $\mathsf{pp} := i\mathcal{O}(\mathsf{prog}_1[K_1\{x_1,x_2\}, K_2\{x_1,x_2\}, K_3\{x_1,x_2\}, s_0, s_1, h_0, h_1, Z_b, Z_{1-b}, \lambda, t = t(\lambda)])$.

Intuitively, indistinguishability follows from (g, ε_{MHP})-security of the underlying memory hard puzzle MHP. However, we stress that indistinguishability only followed against a aAT bounded adversary who is not able to win the MHP security game. For example, if $b = 1$ and attacker is able to solve $Z_0 := \mathsf{pp}_{H_3}(x_0, \varnothing)$ then the attacker might notice that the order of the hardcoded puzzles Z_b and Z_{1-b} was swapped in comparison to the solutions s_0 and s_1 which will never happen in hybrid 2. We argue that if the attacker can distinguish between hybrids 2 and 3 then we can simulate the attacker to win the MHP security game. It follows that any attacker $\mathcal{A}$ with bounded $\mathsf{aAT}(\mathcal{A})$ cannot distinguish between hybrids H_2 and H_3.

Finally, we remark that an MHF attacker has negigible advantage in hybrid H_3. Otherwise, we could break security of the underlying MHP since the puzzles Z_0 and Z_1 are presented in random order. For formal proof we refer to Appendix D.2.

D.2 Indistinguishability of Hybrid 2 and 3

It remains to argue that hybrids H_2 and H_3 are indistinguishable.

Lemma D.1 (Indistinguishability of hybrid H_2 and H_3)**.** *Suppose that a (g, ε_{MHP})-MHP is used in Construction 5.3. Then, for any distinguisher $\mathcal{A}$ with $\mathsf{aAT}(\mathcal{A}) \leqslant y$ for the function $y(\lambda) = g(t(\lambda), \lambda)/p(\log t(\lambda), \lambda)^2$ and any $\lambda > 0$ we have*

$$\left|\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_3}) = 1] - \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_2}) = 1]\right| \leqslant \varepsilon_{MHP}(\lambda) .$$

Here, $p(\cdot,\cdot)$ is a fixed polynomial which depends on the efficiency of the underlying MHP and $i\mathcal{O}$ constructions.

Proof. To prove this lemma, we first suppose for contradiction that there exists an adversary, say $\mathcal{A}$, who can distinguish between hybrids H_2 and H_3 with advantage $f(\lambda) > \varepsilon_{MHP}(\lambda)$. Then, we will construct another adversary $\mathcal{B}$ with $\mathsf{aAT}(\mathcal{B}, \lambda) < \mathsf{aAT}(\mathcal{A}, \lambda) \cdot p(\log t(\lambda), \lambda)^2 \leqslant g(t(\lambda), \lambda)$ who simulates $\mathcal{A}$ to break (g, ε_{MHP})-security for the underlying MHP.

Our MHP attacker $\mathcal{B}(Z_b, Z_{1-b}, s_0, s_1)$ attempts to solve its MHP challenge (Z_b, Z_{1-b}, s_0, s_1) as follows: First, $\mathcal{B}$ sets $\mathsf{pp} \leftarrow i\mathcal{O}(\mathsf{prog}_1[K_1\{x_0,x_1\}, K_2\{x_0,x_1\}, K_3\{x_0,x_1\}, s_0, s_1, h_0, h_1, Z_b, Z_{1-b}, \lambda, t(\lambda)])$ where h_0 and h_1 are selected uniformly at random. Then the adversary $\mathcal{B}$ runs $\mathcal{A}(x_0, x_1, \mathsf{pp})$ to obtain a bit b', and outputs b'.

Analysis: Observe that pp is generated exactly as in hybrid H_3. Conditioning on the event that $b = 0$ we have that pp is generated as in hybrid H_2. Thus, we have $\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) = 1 | b = 0] = \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_2}) = 1]$ and

$$\begin{aligned}\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) = 1 | b = 1] &= 2\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) = 1] - \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) | b = 0] \\ &= 2\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_3}) = 1] - \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_2}) = 1] .\end{aligned}$$

We note that $\mathcal{B}$ wins with probability

$$\begin{aligned}\Pr[\mathcal{B}(s_0, s_1, Z_b, Z_{1-b}) = b] &= \frac{1}{2}\left(\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) = 1 | b = 1]\right) + \frac{1}{2}\left(1 - \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}) = 1 | b = 0]\right) \\ &= \Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_3}) = 1] - \frac{1}{2}\Pr[\mathcal{A}(x_0, x_1, \mathsf{pp}_{H_2}) = 1]\end{aligned}$$

$$-\frac{1}{2}\Pr[\mathcal{A}(x_0,x_1,\mathsf{pp})=1|b=0]+\frac{1}{2}$$
$$=\Pr[\mathcal{A}(x_0,x_1,\mathsf{pp}_{H_3})=1]-\Pr[\mathcal{A}(x_0,x_1,\mathsf{pp}_{H_2})=1]+\frac{1}{2}.$$

So we have

$$\left|\Pr[\mathcal{B}(s_0,s_1,Z_b,Z_{1-b})=b]-\frac{1}{2}\right|=\left|\Pr[\mathcal{A}(x_0,x_1,\mathsf{pp}_{H_3})=1]-\Pr[\mathcal{A}(x_0,x_1,\mathsf{pp}_{H_2})=1]\right|\geqslant f(\lambda).$$

This contradicts the security of the underlying MHP as long as the aAT complexity of $\mathcal{B}$ is sufficiently small; i.e., $\mathsf{aAT}(\mathcal{B},\lambda)<g(t'(\lambda),\lambda)$.

Finally, we analyze the aAT cost of $\mathcal{B}$. Note that a circuit for $\mathcal{B}_\lambda$ requires *at most* $p(\log(t'),\lambda)$ additional gates to generate $\mathsf{pp}\leftarrow i\mathcal{O}(\mathsf{prog}_1[K_1\{x_0,x_1\},K_2\{x_0,x_1\},K_3\{x_0,x_1\},s_0,s_1,h_0,h_1,Z_b,Z_{1-b},\lambda,t(\lambda)])$ before simulating $\mathcal{A}$. Here, the specific polynomial $p(\cdot,\cdot)$ depends on the complexity of the underlying $i\mathcal{O}$ construction and the underlying MHP construction. Suppose that the circuit for $\mathcal{A}_\lambda$ had depth $d>1$ (time) and G gates (area) then the circuit for $\mathcal{B}_\lambda$ would have depth at most $d+p(\log(t'),\lambda)$ and at most $G+p(\log(t'),\lambda)$ gates. The aAT complexity of $\mathcal{B}_\lambda$ would be at most

$$\begin{aligned}(d+p(\log(t'),\lambda))\cdot(G+p(\log(t'),\lambda)) &\leqslant G\cdot d+(G+d)\cdot p(\log(t'),\lambda)+p(\log(t'),\lambda)^2\\ &\leqslant G\cdot d\cdot(1+p(\log(t'),\lambda))+p(\log(t'),\lambda)^2\\ &\leqslant G\cdot d\cdot p(\log(t'),\lambda)^2\\ &\leqslant \mathsf{aAT}(\mathcal{A},\lambda)\cdot p(\log(t'),\lambda)^2=g(t(\lambda),\lambda).\end{aligned}$$

□

Lemma D.2 (Bounded advantage in H_3)**.** *Suppose that we use a (g,ε_{MHP})-secure MHP and $(t_{i\mathcal{O}},\varepsilon_{i\mathcal{O}})$-secure $i\mathcal{O}$ in Construction 5.3. Then, for any $\mathcal{A}$ with $\mathsf{aAT}(\mathcal{A})\leqslant y$ with $y(\lambda)=g(t(\lambda),\lambda)/p(\log(t),\lambda)^2$ and any $\lambda>\lambda_0$ we have*

$$\left|\Pr[\mathcal{A}(x_0,h_b,\mathsf{pp}_{H_3})=b]-\frac{1}{2}\right|\leqslant\varepsilon_{MHP}(\lambda)\ ,$$

where the specific polynomial $p(\cdot,\cdot)$ depends on the efficiency of the underlying constructions of $i\mathcal{O}$ and the memory-hard puzzle.

Proof. Assume by contradiction that an MHF attacker $\mathcal{A}$ wins the MHF security game with advantage $f(\lambda)>\varepsilon_{MHP}(\lambda)$. We define a MHP attacker $\mathcal{B}(Z_b,Z_{1-b},s_0,s_1)$ as follows: first we generate pp as $\mathsf{pp}\leftarrow i\mathcal{O}(\mathsf{prog}_1[K_{j\in[3]}\{x_0,x_1\},s_0,s_1,h_0,h_1,Z_b,Z_{1-b},\lambda,t(\lambda)])$ where the values s_i,h_i,r_i are sampled randomly and $Z_i=\mathsf{Puz.Gen}(1^\lambda,s_i;r_i)$. Next we run $\mathcal{A}(x_0,h_0,\mathsf{pp})$ to obtain a bit b' which we output. Observe that $\mathcal{B}$'s advantage is identical to that of $\mathcal{A}$; i.e., $f(\lambda)$. In particular, if $b=0$ then $\Pr[b'=b|b=0]=\Pr[\mathcal{A}(x_0,h_0,\mathsf{pp}_{H_3})=0]$. Similarly, $\Pr[b'=b|b=1]=\Pr[\mathcal{A}(x_0,h_1,\mathsf{pp}_{H_3})=1]$ since swapping Z_0,Z_1 is equivalent to swapping h_0,h_1. Thus we have

$$\left|\Pr[b=b']-\frac{1}{2}\right|=f(\lambda)\ .$$

Thus, if $\mathcal{B}$'s aAT complexity is sufficiently small we obtain a contradiction. As in the proof of Lemma D.1 above the aAT complexity increases by a multiplicative factor of $p(t(\lambda),\lambda)^2$ at worst where the specific polynomial $p(\cdot,\cdot)$ depends on the complexity of the underlying $i\mathcal{O}$ construction and the underlying MHP construction. □

We are now ready to prove Theorem 5.4.

Proof of Theorem 5.4. Fix our MHF attacker $\mathcal{A}(x_0, h_b)$ with $\mathsf{aAT}(\mathcal{A}) \leqslant y$ with $y(\lambda) = g(t(\lambda), \lambda)/p(\log(t), \lambda)^2$. $\mathcal{A}$ attempts to distinguish h_b if a real value or a uniformly random value. From Lemma D.2 we have

$$\left|\Pr[\mathcal{A}(x_0, h_b, \mathsf{pp}_{H_3}) = b] - \frac{1}{2}\right| \leqslant \varepsilon_{MHP}(\lambda) \ .$$

Since hybrids H_2 and H_3 are indistinguishable we can apply Lemma D.1 to show that

$$\left|\Pr[\mathcal{A}(x_0, h_b, \mathsf{pp}_{H_2}) = b] - \frac{1}{2}\right| \leqslant \left|\Pr[\mathcal{A}(x_0, h_b, \mathsf{pp}_{H_3}) = b] - \frac{1}{2}\right| + \varepsilon_{MHP} \leqslant 2\varepsilon_{MHP}(\lambda) \ .$$

Note that since $\mathcal{A}$ has $\mathsf{aAT}(\mathcal{A}, \lambda) \leqslant g(t(\lambda), \lambda)/p(\log t, \lambda)^2$ we can assume that $\mathcal{A}$ runs in time less than $\min\{t_{i\mathcal{O}}(\lambda), t_{PPRF}(\lambda)\}$. Thus, by PPRF security we have

$$\begin{aligned}\left|\Pr[\mathcal{A}(x_0, x_1, h_b, h_{1-b}, \mathsf{pp}_{H_1}) = b] - \frac{1}{2}\right| &\leqslant \left|\Pr[\mathcal{A}(x_0, x_1, h_b, h_{1-b}, \mathsf{pp}_{H_2}) = b] - \frac{1}{2}\right| + 3\varepsilon_{PPRF}(\lambda) \\ &\leqslant 3\varepsilon_{PPRF}(\lambda) + 2\varepsilon_{MHP}(\lambda) \ .\end{aligned}$$

Finally, by $i\mathcal{O}$ security we have

$$\begin{aligned}\left|\Pr[\mathcal{A}(x_0, h_b, \mathsf{pp}_{H_0}) = b] - \frac{1}{2}\right| &\leqslant \left|\Pr[\mathcal{A}(x_0, x_1, h_b, h_{1-b}, \mathsf{pp}_{H_1}) = b] - \frac{1}{2}\right| + \varepsilon_{i\mathcal{O}}(\lambda) \\ &\leqslant \varepsilon_{i\mathcal{O}}(\lambda) + 3\varepsilon_{PPRF}(\lambda) + 2\varepsilon_{MHP}(\lambda) \ .\end{aligned}$$

□

E Proof of Theorem 6.8

We first recall Theorem 6.8 and Construction 6.7.

Theorem 6.8. *Let $\mathbb{C}$ be a class of algorithms. Let $C_\mathsf{p}[K_\mathsf{p}, k_\mathsf{p}, \lambda]$ be a $(\ell_\mathsf{p}, \delta_\mathsf{p}, p_\mathsf{p}, \varepsilon_\mathsf{p})$-private* LDC *and let $C_*[K_*, k_*]$ be a (ℓ_*, δ_*, p_*)-*LDC^**. Further assume that Enc_p, Dec_p, and Enc_* are contained in $\mathbb{C}$. If there exists a $(\mathbb{C}, \varepsilon')$-hard puzzle, then Construction 6.7 is a $(\ell, \delta, p, \varepsilon, \mathbb{C})$-locally decodable code $C[K, k] = (\mathsf{Enc}, \mathsf{Dec})$ with $k = k_\mathsf{p}$, $K = K_\mathsf{p} + K_*$, $\ell = \ell_\mathsf{p} + \ell_*$, $\delta = (1/K) \cdot \min\{\delta_* \cdot K_*, \delta_\mathsf{p} \cdot K_\mathsf{p}\}$, $p \geqslant 1 - k_\mathsf{p}(2 - p_\mathsf{p} - p_*)$, and $\varepsilon = k \cdot (\varepsilon_\mathsf{p} \cdot p + 2\varepsilon')/(1 - p)$.*

Construction 6.7. *Let $C_\mathsf{p}[K_\mathsf{p}, k_\mathsf{p}, \lambda] = (\mathsf{Gen}, \mathsf{Enc}_\mathsf{p}, \mathsf{Dec}_\mathsf{p})$ be a private* LDC*, let $C_*[K_*, k_*] = (\mathsf{Enc}_*, \mathsf{Dec}_*)$ be a* LDC^**, and let $\mathsf{Puz} = (\mathsf{Puz.Gen}, \mathsf{Puz.Sol})$ be a $(\mathbb{C}, \varepsilon')$-hard puzzle. Let t' be the polynomial guaranteed by Definition 6.5. Then we construct $C[K, k] = (\mathsf{Enc}, \mathsf{Dec})$ as follows:*

$\mathsf{Enc}(x,\lambda)[C_\mathsf{p}, C_*, \mathsf{Puz}]$:	$\mathsf{Dec}^{Y'_\mathsf{p} \circ Y'_*}(i,\lambda)[C_\mathsf{p}, C_*, \mathsf{Puz}]$:
1. *Sample random seed* $s \xleftarrow{\$} \{0,1\}^{k_\mathsf{p}}$.	1. *Decode* $Z \leftarrow \mathsf{Dec}_*^{Y'_*}$.
2. *Choose polynomial* $t > t'$ *and compute* $Z \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s)$, *where* $Z \in \{0,1\}^{k_*}$.	2. *Compute* $s \leftarrow \mathsf{Puz.Sol}(Z)$.
3. *Set* $Y_* \leftarrow \mathsf{Enc}_*(Z)$.	3. *Compute* $\mathsf{sk} \leftarrow \mathsf{Gen}_\mathsf{p}(1^\lambda; s)$.
4. *Set* $\mathsf{sk} \leftarrow \mathsf{Gen}_\mathsf{p}(1^\lambda; s)$.	4. *Output* $\mathsf{Dec}_\mathsf{p}^{Y'_\mathsf{p}}(i; \mathsf{sk})$.
5. *Set* $Y_\mathsf{p} \leftarrow \mathsf{Enc}_\mathsf{p}(x,\lambda;\mathsf{sk})$.	
6. *Output* $Y_\mathsf{p} \circ Y_*$.	

Proof. We first remark that definitions of k, K, ℓ, δ, and p follow directly by construction. We now turn to arguing the security of our scheme under the game `LDC-Sec-Game`, which we recall next.

`LDC-Sec-Game`$(\mathcal{A}, x, \lambda, \delta, p)$:

1. The challenger computes $Y \leftarrow \mathsf{Enc}(x,\lambda)$ encoding the message x and sends $Y \in \{0,1\}^K$ to the attacker.

2. The channel $\mathcal{A}$ outputs a corrupted codeword $Y' \leftarrow \mathcal{A}(x, Y, \lambda, \delta, p, k, K)$ where $Y' \in \{0,1\}^K$ has Hamming distance at most δK from Y.

3. The output of the experiment is determined as follows:

$$\texttt{LDC-Sec-Game}(\mathcal{A}, x, \lambda, \delta, p) = \begin{cases} 1 & \text{if } \mathsf{HAM}(Y, Y') \leqslant \delta K \text{ and } \exists i \leqslant k \text{ such that } \Pr[\mathsf{Dec}^{y'}(i,\lambda) = x_i] < p \\ 0 & \text{otherwise} \end{cases}$$

   If the output of the experiment is 1 (resp. 0), the channel is said to *win* (resp. *lose*).

To prove security, we assume that if there exists an adversary $\mathcal{A} \in \mathbb{C}$ that, given the puzzle Puz, can win `LDC-Sec-Game` with probability at least ε, then we can construct an adversary $\mathcal{B} \in \mathbb{C}$ which breaks the $(\mathbb{C}, \varepsilon')$-hard puzzle.

To prove this, we employ a two-phase hybrid distinguishing argument. We first phase defines two encoders: Enc_0 and Enc_1. The encoder Enc_0 is exactly identical to the encoding function of Construction 6.7, which we denote as Enc. The encoder Enc_1 is our hybrid encoder, and is defined as follows.

$\mathsf{Enc}_1(x, \lambda, \mathsf{sk})$:

1. Sample $s' \xleftarrow{\$} \{0,1\}^{k_\mathsf{p}}$.

2. Choose polynomial $t > t'$ and compute $Z' \leftarrow \mathsf{Puz.Gen}(1^\lambda, t(\lambda), s')$.

3. Set $Y_* \leftarrow \mathsf{Enc}_*(Z')$.

4. Set $Y_\mathsf{p} \leftarrow \mathsf{Enc}_\mathsf{p}(x, \lambda; \mathsf{sk})$.

5. Output $Y_\mathsf{p} \circ Y_*$.

Given the encoders $\mathsf{Enc}_0, \mathsf{Enc}_1$, we construct our two-phase hybrid distinguisher $\mathcal{D} = (\mathcal{D}_1, \mathcal{D}_2)$ as follows. Phase one consists of the algorithm algorithm $\mathcal{D}_1$ which is given as input message x, and is additionally given access to both Enc_0 and Enc_1. Then $\mathcal{D}_1$ performs the following computations:

1. flips bit $b \xleftarrow{\$} \{0,1\}$; and
2. outputs codeword $Y_b \leftarrow \mathsf{Enc}_b(x, \lambda, \mathsf{sk}_b)$.

The output of $\mathcal{D}_1(x)$ is then given to the adversarial channel, resulting in $Y'_b = Y'_{\mathsf{p},b} \circ Y'_* \leftarrow \mathcal{A}(Y_b)$ for $\mathcal{A} \in \mathbb{C}$. Here, $Y'_{\mathsf{p},b}$ is a substring of Y'_b that corresponds to the corruption of the codeword $Y_{\mathsf{p},b} \leftarrow \mathsf{Enc}_{\mathsf{p}}(x, \mathsf{sk}_b)$. Phase two consists of the algorithm $\mathcal{D}_2$ which is given as input the original message x, the secret key sk_b, and the corrupt codeword $Y'_{\mathsf{p},b}$, where b corresponds to the bit that was flipped when running $\mathcal{D}_1(x)$. Upon this input, $\mathcal{D}_2$ performs the following computations:

1. sample $i \xleftarrow{\$} [|x|]$;
2. run $x'_i \leftarrow \mathsf{Dec}_{\mathsf{p}}^{Y'_{\mathsf{p},b}}(i; \mathsf{sk}_b)$; and
3. output $b' = 0$ if $x_i \neq x'_i$; otherwise output $b' = 1$.

We say that the distinguisher $\mathcal{D}$ wins if $b = b'$.

We formally give our two-phase distinguisher which breaks the $(\mathbb{C}, \varepsilon')$-hard puzzle if there exists a channel $\mathcal{A} \in \mathbb{C}$ which wins `LDC-Sec-Game` with probability at least ε. Suppose such an adversary $\mathcal{A}$ exists. For puzzle solutions s_0, s_1 (viewed as independent random strings), we want to construct an adversary $\mathcal{B} \in \mathbb{C}$ which distinguishes (Z_b, Z_{1-b}, s_0, s_1) with probability at least ε' for $b \xleftarrow{\$} \{0,1\}$. Fix a message x and security parameter λ. Our adversary $\mathcal{B}$ is constructed as follows: suppose $\mathcal{B}$ is given as input (Z_b, Z_{1-b}, s_0, s_1) for some $b \xleftarrow{\$} \{0,1\}$ unknown to $\mathcal{B}$.

1. Fix message x.
2. Encode the message x as follows:
   (a) Obtain $\mathsf{sk} \leftarrow \mathsf{Gen}_{\mathsf{p}}(1^\lambda, s_0)$.
   (b) Set $Y_* \leftarrow \mathsf{Enc}_*(Z_b)$.
   (c) Set $Y_{\mathsf{p}} \leftarrow \mathsf{Enc}_{\mathsf{p}}(x, \lambda; \mathsf{sk})$.
   (d) Set $Y = Y_{\mathsf{p}} \circ Y_*$.
3. Obtain $Y' \leftarrow \mathcal{A}(x, Y, \lambda, \delta, p, k, K)$.
4. Set Y'_{p} to be the substring of Y' that corresponds to the corruption of Y_{p} above.
5. Simulate $x'_i \leftarrow \mathsf{Dec}_{\mathsf{p}}^{Y'_{\mathsf{p}}}(i, \mathsf{sk})$ for $i \xleftarrow{\$} [|x|]$.
6. If $x_i \neq x'_i$ output $b' = 0$. Else output $b' = 1$.

We first note that by assumption, $\mathcal{B} \in \mathbb{C}$ since $\mathcal{A}, \mathsf{Enc}_{\mathsf{p}}, \mathsf{Dec}_{\mathsf{p}}, \mathsf{Enc}_* \in \mathbb{C}$. Now we argue that our adversary distinguishes (Z_b, Z_{1-b}, s_0, s_1) with noticeable probability. First note that sk is always generated as $\mathsf{Gen}_{\mathsf{p}}(1^\lambda, s_0)$. Notice that for $b = 1$ the puzzle Z_1 is encoded as Y_*, and the secret key sk is unrelated to the solution s_1 of puzzle Z_1. In this case, the adversary $\mathcal{A}$ wins the `LDC-Sec-Game` with probability at most ε_{p}; this holds information theoretically since sk and Y_* are completely unrelated and uncorrelated. In particular, with probability at most ε_{p}, $\mathcal{A}$ introduces an error pattern such that the distance between Y and Y' is at most δK and there exists $i \leqslant k$ such that the decoder outputs x_i with probability less than p. For the case $b = 0$, puzzle Z_0 is encoded as Y_* and has solution s_0, which is used to generate sk. Thus in this case, the probability that the decoder outputs an incorrect x_i for some $i \leqslant k$ with at most probability p is at least ε since we assume $\mathcal{A}$ wins `LDC-Sec-Game` with probability at least ε.

We analyze the probability $\mathcal{B}$ outputs bit b'. First consider the case where $b = 0$. Then the probability that $b' = 0$ is at least $\varepsilon \cdot (1-p) \cdot (1/k)$ by the argument above. Now for $b = 1$, the probability that $b' = 0$ is at most $\varepsilon_{\mathsf{p}} \cdot p$, which implies that $b' = 1$ is at least $1 - \varepsilon_{\mathsf{p}} \cdot p$. Therefore

$$\Pr_{b \xleftarrow{\$} \{0,1\}} [\mathcal{B}(Z_b, Z_{1-b}, s_0, s_1) = b] \geqslant \frac{1}{2}(\varepsilon \cdot (1-p) \cdot (1/k) + 1 - \varepsilon_{\mathsf{p}} \cdot p)$$

which implies that

$$\Pr_{b \xleftarrow{\$} \{0,1\}} [\mathcal{B}(Z_b, Z_{1-b}, s_0, s_1) = b] - \frac{1}{2} \geqslant \frac{\varepsilon \cdot (1-p)\dot{(}1/k) - \varepsilon_{\mathsf{p}} \cdot p}{2} = \varepsilon'.$$

Thus $\mathcal{B}$ breaks Puz with probability at least ε', which contradicts the hardness of Puz. ☐

F Powers of 2 Graph Proofs

F.1 Proof of Lemma 7.2

The results below have been known to the MHF research community for several years, but to the best of our knowledge have never been published. Thus, we emphasize that we do not claim credit for this result or view this result as an original contribution of our paper. However, we include the proof below for completeness since we cannot cite a concrete source for validation.

Definition F.1. *Fix a graph $G = (V, E)$ on N vertices and without loss of generality let $V = [N]$. We say that a node $u \in V$ is* α-good *with respect to a set $S \subset V$ of deleted nodes if for all r the intervals $F_r(u) = [u - r + 1, u]$ and $B_r(u) = [u - r + 1, u]$ both contain at most $\alpha \cdot |F_r(u)|$ and $\alpha \cdot |B_r(u)|$ nodes in S, respectively.*

The proof follows by the following series of claims. Fix N and let $G := G_{\mathsf{Po2}}$ be the powers of two graph defined in Section 7.1 on N vertices.

Claim F.2 ([EGS75, ABP18]). *If $|S| = e = N/(16 \cdot \log(N))$ then at least $N - 2e/\alpha = N/2$ nodes are α-good with respect to S.*

Claim F.3. *For all $i \geqslant 0$, at least $(1 - i/\alpha)$-fraction of nodes in $[u, u - 1 + 2^i]$ are reachable from u in $G - S$.*

Proof. Clearly this holds for $i = 0$ or $i = 1$. Suppose that at most x-fraction of nodes in $A_i = [u, u-1+2^i]$ are not reachable from u. Consider the set $B_i = [u + 2^i, u^{+}2^{i+1} - 1]$. Notice that if $v \in A$ is reachable from u then the node $v' = v + 2^i$ is also reachable from u as long as v' has not been deleted—since the edge $(v, v + 2^i)$ must exist by definition of the graph. Since u is α-good, at most 2α-fraction of the nodes (i.e., at most $\alpha 2^{i+1}$ nodes) can be deleted from B. Thus at most $(2\alpha + x)$-fraction of nodes in B are not reachable from u and at most $(\alpha + x)$-fraction are not reachable in $A + B = [u, 2^{i+1} - 1]$. ☐

Claim F.4. *If u is $\alpha = 1/(8\log(N))$ good with respect to S then for any $r > 0$ at least $(3/4)$-fraction of nodes in $[u, u + r - 1]$ are reachable from u in $G - S$.*

Proof. Follows immediately by Claim F.3. ☐

Claim F.5. *If v is $\alpha = 1/(8 \log N)$ good with respect to S then for any $r > 0$ the node v is reachable from at least $(3/4)$ fraction of nodes in $[v - r + 1, v]$ in $G - S$.*

Proof. Symmetric reasoning as above. ☐

Claim F.6. *Suppose that nodes u and v are both $\alpha = 1/(8\log(N))$ good with respect to S. Then there is a path between u and v.*

Proof. Suppose that $u < v$. If $v = u + 1$ then this is immediate. Otherwise by the Pigeonhole Principle, there must be an intermediate node w ($u < w < v$) which is reachable from u (Claim F.4), and from which v is reachable (Claim F.5). It follows that there is a path from u to v through w. $\square$

Claim F.7. *G is (e, d)-depth robust with $e = N/(16\log N)$ and $d = N/2$, and has cumulative pebbling complexity at least $N^2/(32\log N)$.*

Proof. For any set $S \subset V$ of size at most $e = N/(16\log(N))$ there are at least $d = N/2$ nodes that are $\alpha = 1/(8\log(N))$ good with respect to S by Claim F.2. By Claim F.6, there is a directed path which contains all of these α-good nodes. $\square$

F.2 Proof of Proposition 7.4

Proposition 7.4. *Let $N, \lambda \in \mathbb{N}$. Let $\mathcal{L}^{\lambda}_{\mathsf{Po2},N}$ be the language for the relation $\mathcal{R}_{\mathsf{Po2},N}$ instantiated with $x, y \in \{0,1\}^{\lambda}$ and hash function $H_{N,\lambda}\colon \{0,1\}^{\lambda\log(N)} \to \{0,1\}^{\lambda}$ such that $H_{N,\lambda}$ is a uniformly succinct circuit of size $N \cdot \mathrm{poly}(\lambda, \log(N))$. Then $\mathcal{L}^{\lambda}_{\mathsf{Po2},N} \in \mathsf{SC}_{N'}$ for $N' = N^2$.*

Proof. It suffices to prove that there exists a circuit a uniformly succinctly describable circuit which computes $F^{H}_{G_{\mathsf{Po2},N}}$ of size $O(N' \cdot \mathrm{poly}(\lambda, \log(N')))$. In particular, we construct a circuit $C_{N,\lambda}$ which computes $F_{G_{\mathsf{Po2}},H}$ of size $O(N \cdot \mathrm{poly}(\lambda, \log(N)))$ that is succinctly describable by a circuit of size $O(\mathrm{polylog}(\lambda, N))$.

Construction of $C_{N',\lambda}$ is clear; namely, it is a layered circuit of repeated applications of $H_{N,\lambda}$ where the inputs are specified by different output layers. Recall that by the definition of $G_{\mathsf{Po2},N}$, we have that for $v \in V$ if $v = 1$ then $L_v = H_{N,\lambda}(x)$, where the input is padded with 0's whenever it is less than $\lambda \cdot \log(N)$ bits. Otherwise for $v > 1$ and $n = \log(N)$ we have $L_v = H_{N,\lambda}(L_{u_1}, L_{u_2}, \ldots, L_{u_{n-1}})$, where $u_i = v - 2^i$ if $2^i < v$ and 0 otherwise. Consider block $i \in [n]$ of the input to $H_{N,\lambda}$. The bit $j \in [\lambda]$ of block i corresponds to bit j of the label L_{u_i}, where $L_0 := 0^{\lambda}$ is hardcoded. Thus for fixed N and λ it is simple to construct the circuit $C_{N,\lambda}$ by the process described above. Moreover, we have N applications of the function $H_{N,\lambda}$, which is assumed to have size $O(N \cdot \mathrm{poly}(\lambda, \log(N)))$, which implies our desired size of $O(N' \cdot \mathrm{poly}(\lambda, \log(N')))$.

To obtain a succinct circuit of size $O(\mathrm{polylog}(\lambda, N'))$ which represents $C_{N,\lambda}$, we use the succinct representation of $H_{N,\lambda}$ along with the above logic for input wires to a particular layer i. In particular, let H_{sc} be the uniform succinct circuit of size $O(\mathrm{polylog}(\lambda, N))$ describing $H_{N,\lambda}$. Suppose that the exact size of H_{sc} is S. Then on input $g \in [\lambda\log(N) + 1, S - \lambda]$ the function $H_{\mathsf{sc}}(g) = (i, j, f_g)$ where i and j are the labels of the parent of g and f_g is the functionality computed by gate g. We define a function which succinctly represents a layer i as follows. On input $g \in [\lambda \cdot \log(N) + 1, N - S]$, the function ℓ_i outputs a tuple $((i_1, j_1), (i_2, j_2), f_g)$ such that gate g has parent gates $j_1, j_2 \in [\lambda\log N + 1, N - S]$ in layers $i_1, i_2 \in [0, N]$, respectively, noting that layer 0 is a layer of hardcoded 0's.

More formally, for $g \in [\lambda \cdot \log(N) + 1, S - \lambda]$, $\ell_i(g)$ computes the following.

1. Set $(j_1, j_2, f_g) = H_{\mathsf{sc}}(g)$.
2. If $j_1 \in [\lambda\log(N)]$ (i.e., it is an input gate):
   (a) Let $k \in [\log(N)]$ be such that $(k-1)\lambda + 1 \leqslant j_1 \leqslant k\lambda$, and let $\alpha = j_1 - (k-1)\lambda$.
   (b) If $i_1 = i - 2^{k-1} > 0$, then set j'_1 to be output wire α of layer i_1. Otherwise if $i' \leqslant 0$ set j'_1 to be $(0, 0)$.
3. If $j_2 \in [\lambda \cdot \log(N)]$ then perform the same steps as above, obtaining i_2, j'_2.

4. Output $((i_1, j'_1), (i_2, j'_2))$.

Then the function which describes the whole circuit is $\ell(i, g) := \ell_i(g)$.[9] We claim that the size of $\ell(i, g)$ is $O(\text{polylog}(\lambda, N))$ by the following observations:

- $\ell(i, g)$ runs circuit H_{sc} which has size $O(\text{polylog}(\lambda, N))$;
- computation of α can be done by a size $O(\text{polylog}(\lambda, N))$ circuit; and
- checking if $i_1, i_2 > 0$ can be done via $k - 1$ shift operations, which can be done by a size $O(\text{polylog}(N))$ circuit.

Finally, we argue that there exists a Turing machine M such that on input $1^\lambda, 1^N$ outputs the circuit $C_{N,\lambda}$. First note by assumption we have that $H_{N,\lambda} \in \mathsf{SC}_{N,\lambda}$, so there exists machine M^H that on input $1^\lambda, 1^N$ outputs the succinct circuit for $H_{N,\lambda}$. Next it's clear that the circuits for computing α and checking i_1, i_2 are easily describable by a machine M on input $1^\lambda, 1^N$. Thus we can construct a machine M' which runs machine M^H and M to obtain succinct circuit $C_{N,\lambda}$. Thus $\mathcal{L}^\lambda_{\mathsf{Po2},N} \in \mathsf{SC}_{N'}$. $\square$

G Turing Machine Simulation Proofs

G.1 Proof of Lemma 8.4

For all $i < k < j$ we can reconstruct the content and state of cell k. So, we start simulating Turing machine from i to recover k. Now, we use the data available in the sates $\texttt{Compress}(i, j, t')$ to compute the k-th cell content. Observing that at time $t' = t^i_1$ we have $T[k, t'] = \Box$ (blank) for every $i < k < j$, we begin the simulation with $(t^i_1, S[t^i_1], T[i, t^i_1]) \in \texttt{Compress}(i, j, t')$ for as long as the tape head stays in the interval $[i, j]$. If during the simulation the head goes to the right of j (resp., left of i), we halt computation and lookup the next time t^j_l (reps., t^i_l) when the Turing machine head moves back to cell j (resp. i) along with the corresponding state $(t^i_l, S[t^i_l], T[i, t^i_l]) \in \texttt{Compress}(i, j, t')$ (resp. $(t^j_l, S[t^j_l], T[j, t^j_l]) \in \texttt{Compress}(i, j, t')$) for some $1 < l \leqslant a$ (reps. $1 < l \leqslant b$). Now, we continue simulation from this new starting point.

By Observation 8.2, for each cell $i \leqslant k \leqslant j$ we have $\chi(k, t') \leqslant \frac{\chi(i,j,t')}{j-i+1}$, and as we have $j - i + 1$ cells, the total time for reconstructing the target cell contents in interval $[i, j]$ is at most $\chi(i, j, t')$. As the size of the this interval is $j - i + 1$, we also need to keep the recovered cells during simulation which results in $O(j - i + 1)$ extra space.

G.2 Proof of Lemma 8.5

This lemma is similar to Lemma 8.4; however, $\texttt{Compress}(i', j', t')$ is not given in advance. So first we need to find some potential cell indices $i - \Delta \leqslant i, j \leqslant j + \Delta$ where $\Delta \in O(j - i)$, and compute $\texttt{Compress}(i', j', t')$. Then the steps are exactly the same as Decompression (Lemma 8.4) and we can recover the contents of tape in the given interval. Therefore, we just need to add and consider these extra space and time costs in our analysis in comparison with the previous lemma.

We define $\Delta = \alpha(j - i)$ for some constant $0 < \alpha < 1$. Then we start simulating the Turing machine for the given interval from cell $i - \Delta$. We simulate this interval twice. For the first time in addition to the cell contents, we also define a counter to store the number of visit we have. The counter requires at most $\log(t)$ bits. We continue computation until the head reaches at cell $j + \Delta$. We set the counter in order to know the number of visits to each cell. Then, we check the cells around i in and interval

[9] We note that this circuit can be transformed to the same syntax as defined in Definition 3.1. We omit this transformation for ease of presentation.

of $[(i-\Delta),(i+\Delta)]$ and find the one whose counter, say $i' = \{k' : \chi(k',t') = \min_{i-\Delta \leqslant k \leqslant i+\Delta} \chi(k,t')\}$. Similarly we do the same for j and determine $j' = \{k' : \chi(k',t') = \min_{j-\Delta \leqslant k \leqslant j+\Delta} \chi(k,t')\}$. Now, we run the simulation for the second time and we store all the visit information at cells $i'.j'$ and basically compute $\texttt{Compress}(i',j',t')$, and remove the contents of other cells. Now given the state $\texttt{Compress}(i'.j',t')$ we can simply follow Lemma 8.4 and recover the cell contents in time $O(j-i)$. We note that, the value of each counter is in fact $\chi(k,t')$ for all $k \in [i,j]$.

Based on the described steps, the running time for traversing the interval for two times is $2\chi(j,i,t')$ based on Observation 8.2 and the fact we simulate twice. This is extra time in comparison with Lemma 8.4, where the running time is also $O(\chi(j,i,t'))$. Therefore, the overall time would be $O(\chi(j,i,t'))$.

For the space usage, we can see that for each cell we need to consider a space for counter which requires $\log t$. In addition, we need to reserve a space for the cells in interval $[(i-\Delta),(i+\Delta)]$ (similarly for $[(j-\Delta),(+\Delta)]$) as the one of them may be selected for compression phase, i..e, $\texttt{Compress}(i',j',t')$. So, based on Observation 8.2 for each i',j' in $\texttt{Compress}(i',j',t')$ we need $\max\{\frac{\chi(i-\Delta,i+\Delta)}{2\Delta},\frac{\chi(j-\Delta,j+\Delta)}{2\Delta}\}$ extra space. Based on the selection of $\Delta = \alpha(j-i)$ we can see that the extra space for this case is at most $\frac{\chi(i,j)}{2(j-i)}$. So the total extra storage is $O(\log(t) + \frac{\chi(i,j)}{(j-i)})$.