Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space

Hamza Abusalah[1], Joël Alwen[1], Bram Cohen[2], Danylo Khilko[3], Krzysztof Pietrzak[1], and Leonid Reyzin[4]

[1] Institute of Science and Technology Austria, habusalah|jalwen|pietrzak@ist.ac.at
[2] Chia Network, bram@chia.network
[3] ENS Paris, dkhilko@ukr.net
[4] Boston University, reyzin@cs.bu.edu

Abstract. Proofs of space (PoS) were suggested as more ecological and economical alternative to proofs of work, which are currently used in blockchain designs like Bitcoin. The existing PoS are based on rather sophisticated graph pebbling lower bounds. Much simpler and in several aspects more efficient schemes based on inverting random functions have been suggested, but they don't give meaningful security guarantees due to existing time-memory trade-offs.

In particular, Hellman showed that any *permutation* over a domain of size N can be inverted in time T by an algorithm that is given S bits of auxiliary information whenever $S \cdot T \approx N$ (e.g. $S = T \approx N^{1/2}$). For *functions* Hellman gives a weaker attack with $S^2 \cdot T \approx N^2$ (e.g., $S = T \approx N^{2/3}$). To prove lower bounds, one considers an adversary who has access to an oracle $f : [N] \to [N]$ and can make T oracle queries. The best known lower bound is $S \cdot T \in \Omega(N)$ and holds for random functions and permutations.

We construct functions that provably require more time and/or space to invert. Specifically, for any constant k we construct a function $[N] \to [N]$ that cannot be inverted unless $S^k \cdot T \in \Omega(N^k)$ (in particular, $S = T \approx N^{k/(k+1)}$). Our construction does not contradict Hellman's time-memory trade-off, because it cannot be efficiently evaluated in forward direction. However, its entire function table can be computed in time quasilinear in N, which is sufficient for the PoS application.

Our simplest construction is built from a random function oracle $g : [N] \times [N] \to [N]$ and a random permutation oracle $f : [N] \to [N]$ and is defined as $h(x) = g(x, x')$ where $f(x) = \pi(f(x'))$ with π being any involution without a fixed point, e.g. flipping all the bits. For this function we prove that any adversary who gets S bits of auxiliary information, makes at most T oracle queries, and inverts h on an ϵ fraction of outputs must satisfy $S^2 \cdot T \in \Omega(\epsilon^2 N^2)$.

1 Introduction

A proof of work (PoW), introduced by Dwork and Naor [DN93], is a proof system in which a prover $\mathcal{P}$ convinces a verifier $\mathcal{V}$ that he spent some computation with

respect to some statement x. A simple PoW can be constructed from a function $H(\cdot)$, where a proof with respect to a statement x is simply a salt s such that $H(s, x)$ starts with t 0's. If H is modelled as a random function, $\mathcal{P}$ must evaluate H on 2^t values (in expectation) before he finds such an s.

The original motivation for PoWs was prevention of email spam and denial of service attacks, but today the by far most important application for PoWs is securing blockchains, most prominently the Bitcoin blockchain, whose security is based on the assumption that the majority of computing power dedicated towards the blockchain comes from honest users. This results in a massive waste of energy and other resources, as this mining is mostly done on dedicated hardware (ASICs) which has no other use than Bitcoin mining. In [DFKP15] proofs of space (PoS) have been suggested as an alternative to PoW. The idea is to use disk space rather than computation as the main resource for mining. As millions of users have a significant amount of unused disk space available (on laptops etc.), dedicating this space towards securing a blockchain would result in almost no waste of resources.

Let $[N]$ denote some domain of size N. For convenience we'll often assume that $N = 2^n$ is a power of 2 and identify $[N]$ with $\{0,1\}^n$, but this is never crucial and $[N]$ can be any other efficiently samplable domain. A simple idea for constructing a PoS is to have the verifier specify a random function $f : [N] \to [N]$ during the initialization phase, and have the prover compute the function table of f and sort it by the output values.[5] Then, during the proof phase, to convince the verifier that he really stores this table, the prover must invert f on a random challenge. We will call this approach "simple PoS"; we will discuss it in more detail in §1.3 below, and explain why it miserably fails to provide any meaningful security guarantees.

Instead, existing PoS [DFKP15,RD16] are based on pebbling lower bounds for graphs. These PoS provide basically the best security guarantees one could hope for: a cheating prover needs $\Theta(N)$ space or time after the challenge is known to make a verifier accept. Unfortunately, compared to the (insecure) simple PoS, they have two drawbacks which make them more difficult to use as replacement for PoW in blockchains. First, the proof size is quite large (several MB instead of a few bytes as in the simple PoS or Bitcoin's PoW). Second, the initialization phase requires two messages: the first message, like in the simple PoS, is sent from the verifier to the prover specifying a random function f, and second message, unlike in the simple PoS, is a "commitment" from the prover to the verifier.[6]

If such a pebbling-based PoS is used as a replacement for PoW in a blockchain design, the first message can be chosen non-interactively by the miner (who plays the role of the prover), but the commitment sent in the second message is more

[5] f must have a short description, so it cannot be actually random. In practice the prover would specify f by, for example, a short random salt s for a cryptographic hash function H, and set $f(x) = H(s, x)$.

[6] Specifically, the prover computes a "graph labelling" of the vertices of a graph (which is specified by the PoS) using f, and then a Merkle tree commitment to this entire labelling, which must be sent back to the verifier.

tricky. In Spacemint (a PoS-based decentralized cryptocurrency [PPK+15]), this is solved by having a miner put this commitment into the blockchain itself before he can start mining. As a consequence, Spacemint lacks the nice property of the Bitcoin blockchain where miners can join the effort by just listening to the network, and only need to speak up once they find a proof and want to add it to the chain.

1.1 Our Results

In this work we "resurrect" the simple approach towards constructing PoS based on inverting random functions. This seems impossible, as Hellman's time-memory trade-offs — which are the reason for this approach to fail — can be generalized to apply to all functions (see Section 1.4). For Hellman's attacks to apply, one needs to be able to evaluate the function efficiently in forward direction. At first glance, this may not seem like a real restriction at all, as inverting functions which cannot be efficiently computed in forward direction is undecidable in general.[7] However, we observe that for functions to be used in the simple PoS outlined above, the requirement of efficient computability can be relaxed in a meaningful way: we only need to be able to compute the entire function table in time linear (or quasilinear) in the size of the input domain. We construct functions satisfying this relaxed condition for which we prove lower bounds on time-memory trade-offs beyond the upper bounds given by Hellman's attacks.

Our most basic construction of such a function $g_f : [N] \to [N]$ is based on a function $g : [N] \times [N] \to [N]$ and a permutation $f : [N] \to [N]$. For the lower bound proof g and f are modelled as truly random, and all parties access them as oracles. The function is now defined as $g_f(x) = g(x, x')$ where $f(x) = \pi(f(x'))$ for any involution π without fixed points. For concreteness we let π simply flip all bits, denoted $f(x) = \overline{f(x')}$. Let us stress that f does not need to be a permutation – it can also be a random function[8] – but we'll state and prove our main result for a permutation as it makes the analysis cleaner. In practice — where one has to instantiate f and g with something efficient — one would rather use a function, because it can be instantiated with a cryptographic hash function like SHA-3 or (truncated) AES,[9] whereas we don't have good candidates for suitable permutations (at the very least f needs to be one-way; and, unfortunately, all

[7] Consider the function $f : \mathbb{N} \times \{0,1\}^* \to \{0,1\} \times \{0,1\}^*$ where $f(s, T) = (b, T)$ with $b = 1$ iff the Turing machine T stops in s steps. Here deciding if $(1, T)$ has a pre-image at all requires to solve the halting problem.

[8] If f is a function, we don't need π; the condition $f(x) = \pi(f(x'))$ can be replaced with simply $f(x) = f(x'), x \neq x'$. Note than now for some x there's no output $g_f(x)$ at all (i.e., if $\forall x' \neq x \ : \ f(x) \neq f(x')$), and for some x there's more than one possible value for $g_f(x)$. This is a bit unnatural, but such a g_f can be used for a PoS in the same way as if f were a permutation.

[9] As a concrete proposal, let $\mathsf{AES}_n : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^n$ denote AES with the output truncated to n bits. We can now define f, g by a random key $k \leftarrow \{0,1\}^{128}$ as $f(x) = \mathsf{AES}_n(k, 0\|x\|0^{128-n-1})$ and $g(x) = \mathsf{AES}_n(k, 1\|x\|0^{128-2n-1})$. As in practice n will be something like $30 - 50$, which corresponds to space (which is $\approx n \cdot 2^n$ bits)

candidates we have for one-way permutations are number-theoretic and thus much less efficient).

In Theorem 2 we state that for g_f as above, any algorithm which has a state of size S (that can arbitrarily depend on g and f), and inverts g_f on an ϵ fraction of outputs, must satisfy $S^2T \in \Omega(\epsilon^2 N^2)$. This must be compared with the best lower bound known for inverting random functions (or permutations) which is $ST = \Omega(\epsilon N)$. We can further push the lower bound to $S^kT \in \Omega(\epsilon^k N^k)$ by

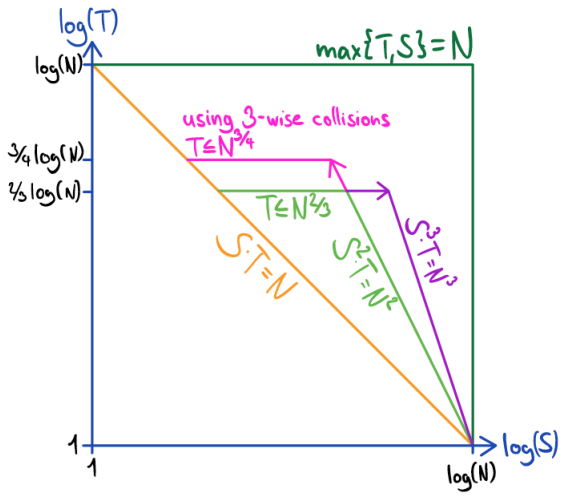

Fig. 1. Illustration of lower bounds. Orange: the $ST = \Omega(N)$ lower bound for inverting random permutations or functions. Dark green: the ideal bound where either T or S is $\Omega(N)$ as achieved by the pebbling-based PoS [DFKP15,RD16] (more precisely, the bound approaches the dark green line for large N). Light green: the lower bound $S^2T = \Omega(N^2)$ for $T \leq N^{2/3}$ for our most basic construction as stated in Theorem 2. Pink: the restriction $T \leq N^{2/3}$ on T we need for our proof to go through can be relaxed to $T \leq N^{t/(t+1)}$ by using t-wise collisions instead of pairwise collisions in our construction. The pink arrow shows how the bound improves by going from $t = 2$ to $t = 3$. Purple: we can push the $S^2T = \Omega(N^2)$ lower bound of the basic construction to $S^kT = \Omega(N^k)$ by using $k-1$ levels of nesting. The purple arrow shows how the bound improves by going from $k = 2$ to $k = 3$.

"nesting" the construction; in the first iteration of this nesting one replaces the inner function f with g_f.[10] These lower bounds are illustrated in Figure 1.

In this paper we won't give a proof for the general construction, as the proof for the general construction doesn't require any new ideas, but just gets

in the Gigabyte to Petabyte range. Using AES with the smallest 128 bit blocksize is sufficient as $2n \ll 128$.

[10] The dream version would be a result showing that one needs either $S = \Omega(N)$ or $T = \Omega(N)$ to invert. Our results approach this as k grows showing that $S = T = \Omega(N^{k/(k+1)})$ is required.

more technical. We also expect the basic construction to be already sufficient for constructing a secure PoS. Although for g_f there exists a time-memory trade-off $S^4T \in O(N^4)$ (say, $S = T \approx N^{4/5}$), which is achieved by "nesting" Hellman's attack,[11] we expect this attack to only be of concern for extremely large N.[12]

A caveat of our lower bound is that it only applies if $T \leq N^{2/3}$. We don't see how to break our lower bound if $T > N^{2/3}$, and the restriction $T \leq N^{2/3}$ seems to be mostly related to the proof technique. One can improve the bound to $T \leq N^{t/(t+1)}$ for any t by generalizing our construction to t-wise collisions. One way to do this – if f is a permutation and t divides N – is as follows: let $g : [N]^t \to [N]$ and define $g_f(x) = g(x, x_1, \ldots, x_{t-1})$ where for some partition $S_1, \ldots, S_{N/t}, |S_i| = t$ of $[N]$ the values $f(x), f(x_1), \ldots, f(x_{t-1})$ contain all elements of a partition S_i and $x_1 < x_2 < \ldots < x_{t-1}$.

1.2 Proofs of Space

A proof of space as defined in [DFKP15] is a two-phase protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, where after an initial phase $\mathcal{P}$ holds a file F of size N,[13] whereas $\mathcal{V}$ only needs to store some small value. The running time of $\mathcal{P}$ during this phase must be at least N as $\mathcal{P}$ has to write down F which is of size N, and we require that it's not much more, quasilinear in N at most. $\mathcal{V}$ on the other hand must be very efficient, in particular, its running time can be polynomial in a security parameter, but must be basically independent of N.

Then there's a proof execution phase — which typically will be executed many times over a period of time — in which $\mathcal{V}$ challenges $\mathcal{P}$ to prove it stored F. The security requirement states that a cheating prover $\tilde{\mathcal{P}}$ who only stores a file F' of size significantly smaller than N either fails to make $\mathcal{V}$ accept, or must invest a significant amount of computation, ideally close to $\mathcal{P}$'s cost during initialization. Note that we cannot hope to make it more expensive than that as a cheating $\tilde{\mathcal{P}}$ can always just store the short communication during initialization, and then reconstruct all of F before the execution phase.

[11] Informally, nesting Hellman's attack works as follows. Note that if we could efficiently evaluate $g_f(.)$, we could use Hellman's attack. Now to evaluate g_f we need to invert f. For this make a Hellman table to invert f, and use this to "semi-efficiently" evaluate $g_f(.)$. More generally, for our construction with nesting parameter k (when the lower bound is $S^kT \in \Omega(N^k)$) the nested Hellman attack applies if $S^{2k}T \in O(N^{2k})$.

[12] The reason is that for this nested attack to work, we need tables which allow to invert with very high probability, and in this case the tables will store many redundant values. So the hidden constant in the $S^4T \in O(N^4)$ bound of the nested attack will be substantial.

[13] We use the same letter N for the space committed by an honest prover in a PoS as we used for the domain size of the functions discussed in the previous section to emphasize that in our construction of a PoS from a function these will be roughly the same. We'll come back to this in Remark 2 in the next section.

1.3 A Simple PoS that Fails

Probably the first candidate for a PoS scheme that comes to mind is to have — during the initalization phase — $\mathcal{V}$ send the (short) description of a "random behaving" function $f : [N] \to [N]$ to $\mathcal{P}$, who then computes the entire function table of f and stores it sorted by the outputs. During proof execution $\mathcal{V}$ will pick a random $x \in [N]$, and then challenge $\mathcal{P}$ to invert f on $y = f(x)$.[14]

An honest prover can answer any challenge y by looking up an entry (x', y) in the table, which is efficient as the table is sorted by the y's. At first one might hope this provides good security against any cheating prover; intuitively, a prover who only stores $\ll N \log N$ bits (i.e., uses space sufficient to only store $\ll N$ output labels of length $\log N$) will not have stored a value $x \in f^{-1}(y)$ for most y's, and thus must invert by brute force which will require $\Theta(N)$ invocations to f. Unfortunately, even if f is modelled as a truly random function, this intuition is totally wrong due to Hellman's time-memory trade-offs, which we'll discuss in the next section.

The goal of this work is to save this elegant and simple approach towards constructing PoS. As discussed before, for our function $g_f : [N] \to [N]$ (defined as $g_f(x) = g(x, x')$ where $f(x) = \overline{f(x')}$) we can prove better lower bounds than for random functions. Instantiating the simple PoS with g_f needs some minor adaptions. $\mathcal{V}$ will send the description of a function $g : [N] \times [N] \to [N]$ and a permutation $f : [N] \to [N]$ to $\mathcal{P}$. Now $\mathcal{P}$ first computes the entire function table of f and sorts it by the output values. Note that with this table $\mathcal{P}$ can efficiently invert f. Then $\mathcal{P}$ computes (and sorts) the function table of g_f (using that $g_f(x) = g(x, f^{-1}(\overline{f(x)}))$. Another issue is that in the execution phase $\mathcal{V}$ can no longer compute a challenge as before – i.e. $y = g_f(x)$ for a random x – as it cannot evaluate g_f. Instead, we let $\mathcal{V}$ just pick a random $y \in [N]$. The prover $\mathcal{P}$ must answer this challenge with a tuple (x, x') s.t. $f(x) = \overline{f(x')}$ and $g(x, x') = y$ (i.e., $g_f(x) = y$). Just sending the preimage x of g_f for y is no longer sufficient, as $\mathcal{V}$ is not able to verify if $g_f(x) = y$ without x'.

Remark 1 (Completeness and Soundness Error). This protocol has a significant soundness and completeness error. On the one hand, a cheating prover $\tilde{\mathcal{P}}$ who only stores, say 10%, of the function table, will still be able to make $\mathcal{V}$ accept in 10% of the cases. On the other hand, even if g_f behaves like a random function, an honest prover $\mathcal{P}$ will only be able to answer a $1 - 1/e$ fraction ($\approx 63\%$) of the challenges $y \in [N]$, as some will simply not have a preimage under g_f.[15]

When used as a replacement for PoW in cryptocurrencies, neither the soundness nor the completeness error are an issue. If this PoS is to be used in a context

[14] Instead of storing all N tuples $(x, f(x))$ (sorted by the 2nd entry), which takes $2N \log N$ bits, one can compress this list by almost a factor 2 using the fact that the 2nd entry is sorted, and another factor $\approx 1 - 1/e \approx 0.632$ by keeping only one entry whenever there are multiple tuples with the same 2nd entry, thus requiring $\approx 0.632N \log N$ bits.

[15] Throwing N balls in N bins at random will leave around N/e bins empty, so g_f's outputs will miss $N/e \approx 0.37 \cdot N$ values in $[N]$.

where one needs negligible soundness and/or completeness, one can use standard repetition tricks to amplify the soundness and completeness, and make the corresponding errors negligible.[16]

Remark 2 (Domain vs. Space). When constructing a PoS from a function with a domain of size N, the space the honest prover requires is around $N \log N$ bits for the simple PoS outlined above (where we store the sorted function table of a function $f : [N] \to [N]$), and roughly twice that for our basic construction (where we store the function tables of $g_f : [N] \to [N]$ and $f : [N] \to [N]$). Thus, for a given amount N' of space the prover wants to commit to, it must use a function with domain $N \approx N'/\log(N')$. In particular, the time-memory trade-offs we can prove on the hardness of inverting the underlying function translate directly to the security of the PoS.

1.4 Hellman's Time-Memory Trade Offs

Hellman [Hel80] showed that any permutation $p : [N] \to [N]$ can be inverted using an algorithm that is given S bits of auxiliary information on p and makes at most T oracle queries to $p(\cdot)$, where ($\tilde{O}$ below hides $\log(N)^{O(1)}$ factors)

$$S \cdot T \in \tilde{O}(N) \quad \text{e.g. when} \quad S = T \approx N^{1/2} \ . \tag{1}$$

Hellman also presents attacks against *random* functions, but with worse parameters. A rigorous bound was only later proven by Fiat and Naor [FN91] where they show that Hellman's attack on *random* functions satisfies

$$S^2 \cdot T \in \tilde{O}(N^2) \quad \text{e.g. when} \quad S = T \approx N^{2/3} \ . \tag{2}$$

Fiat and Naor [FN91] also present an attack with worse parameters which works for *any* (not necessarily random) function, where

$$S^3 \cdot T \in \tilde{O}(N^3) \quad \text{e.g. when} \quad S = T \approx N^{3/4} \ . \tag{3}$$

The attack on a permutation $p : [N] \to [N]$ for a given T is easy to explain: Pick any $x \in [N]$ and define $x_0, x_1, \ldots$ as $x_0 = x$, $x_{i+1} = p(x_i)$, let $\ell \leq N - 1$ be minimal such that $x_0 = x_\ell$. Now store the values $x_T, x_{2T}, \ldots, x_{(\ell \mod T)T}$ in a sorted list. Let us assume for simplicity that $\ell - 1 = N$, so $x_0, \ldots, x_{\ell-1}$ cover the entire domain (if this is not the case, one picks some x' not yet covered and makes a new table for the values $x_0 = x', x_1 = p(x_0), \ldots$). This requires storing $S = N/T$ values. If we have this table, given a challenge y to invert, we just apply p to y until we hit some stored value x_{iT}, then continue applying p to $x_{(i-1)T}$ until

[16] To decrease the soundness error from 0.37 to negligible, the verifier can ask the prover to invert g_f on $t \in \mathbb{N}$ independent random challenges in $[N]$. In expectation g_f will have a preimage on $0.63 \cdot t$ challenges. The probability that – say at least $0.5 \cdot t$ – of the challenges have a preimage is then exponentially (in t) close to 1 by the Chernoff bound. So if we only require the prover to invert half the challenges, the soundness error becomes negligible.

we hit y, at which point we found the inverse $p^{-1}(y)$. By construction this attack requires T invocations to p. The attack on general functions is more complicated and gives worse bounds as we don't have such a nice cycle structure. In a nutshell, one computes several different chains, where for the jth chain we pick some random $h_j : [N] \to [N]$ and compute $x_0, x_1, \ldots, x_n$ as $x_i = f(h_j(x_{i-1}))$. Then, every T'th value of the chain is stored. To invert a challenge y we apply $f(h_1(\cdot))$ sequentially on input y up to T times. If we hit a value x_{iT} we stored in the first chain, we try to invert by applying $f(h_1(\cdot))$ starting with $x_{(i-1)T}$.[17] If we don't succeed, continue with the chains generated by $f(h_2(\cdot)), f(h_3(\cdot)), \ldots$ until the inverse is found or all chains are used up. This attack will be successful with high probability if the chains cover a large fraction of f's output domain.

1.5 Samplability is Sufficient for Hellman's Attack

One reason the lower bound for our function $g_f : [N] \to [N]$ (defined as $g_f(x) = g(x, x')$ where $f(x) = \overline{f(x')}$) does not contradict Hellman's attacks is the fact that g_f cannot be efficiently evaluated in forward direction. One can think of simpler constructions such as $g'_f(x) = g(x, f^{-1}(x))$ which also have this property, but observe that Hellman's attack is easily adapted to break g'_f. More generally, Hellman's attack doesn't require that the function can be efficiently computed in forward direction, it is sufficient to have an algorithm that efficiently samples random input/output tuples of the function. This is possible for g'_f as for a random z the tuple $f(z), g(f(z), z)$ is a valid input/output: $g'_f(f(z)) = g(f(z), f^{-1}(f(z)) = g(f(z), z)$. To adapt Hellman's attack to this setting – where we just have an efficient input/output sampler σ_f for f – replace the $f(h_i(\cdot))$'s in the attack described in the previous section with $\sigma_f(h_i(\cdot))$.

1.6 Lower Bounds

De, Trevisan and Tulsiani [DTT10] (building on work by Yao [Yao90], Gennaro-Trevisan [GT00] and Wee [Wee05]) prove a lower bound for inverting random permutations, and in particular show that Hellman's attack as stated in Eq.(1) is optimal: For any oracle-aided algorithm $\mathcal{A}$, it holds that for most permutations $p : [N] \to [N]$, if $\mathcal{A}$ is given advice (that can arbitrarily depend on p) of size S, makes at most T oracle queries and inverts p on ϵN values, we have $S \cdot T \in \Omega(\epsilon N)$. Their lower bound proof can easily be adapted to random functions $f : [N] \to [N]$, but note that in this case it is no longer tight, i.e., matching Eq.(2). Barkan, Biham, and Shamir [BBS06] show a matching $S^2 \cdot T \in \tilde{\Omega}(N^2)$ lower bound for a restricted class of algorithms.

[17] Unlike for permutations, there's no guarantee we'll be successful, as the challenge might lie on a branch of the function graph different from the one that includes $x_{(i-1)}T$.

1.7 Proof Outline

The starting point of our proof is the $S \cdot T \in \Omega(\epsilon N)$ lower bound for inverting random permutations by De, Trevisan and Tulsiani [DTT10] mentioned in the previous section. We sketch their simple and elegant proof, with a minor adaption to work for functions rather than permutations in Appendix A.

The high level idea of their lower bound proof is as follows: Assume an adversary $\mathcal{A}$ exists, which is given an auxiliary string aux, makes at most T oracle queries and can invert a random permutation $p : [N] \to [N]$ on an ϵ fraction of $[N]$ with high probability (aux can depend arbitrarily on p). One then shows that given (black box access to) $\mathcal{A}_{\mathsf{aux}}(\cdot) \stackrel{\mathsf{def}}{=} \mathcal{A}(\mathsf{aux}, \cdot)$ it's possible to "compress" the description of p from $\log(N!)$ to $\log(N!) - \Delta$ bits for some $\Delta > 0$. As a random permutation is incompressible (formally stated as Fact 1 in §2 below), the Δ bits we saved must come from the auxiliary string given, so $S = |\mathsf{aux}| \gtrapprox \Delta$.

To compress p, one now finds a subset $G \subset [N]$ where (1) $\mathcal{A}$ inverts successfully, i.e., for all $y \in p(G) = \{p(x) \ : \ x \in G\}$ we have $\mathcal{A}^p_{\mathsf{aux}}(y) = p^{-1}(y)$ and (2) $\mathcal{A}$ never makes a query in G, i.e., for all $y \in G$ all oracle queries made by $\mathcal{A}^p_{\mathsf{aux}}(y)$ are in $[N] - G$ (except for the last query which we always assume is $p^{-1}(y)$).

The compression now exploits the fact that one can learn the mapping $G \to p(G)$ given aux, an encoding of the set $p(G)$, and the remaining mapping $[N] - G \to p([N] - G)$. While decoding, one recovers $G \to p(G)$ by invoking $\mathcal{A}^p_{\mathsf{aux}}(\cdot)$ on all values $p(G)$ (answering all oracle queries using $[N] - G \to p([N] - G)$, the first query outside $[N] - G$ will be the right value by construction).

Thus, we compressed by not encoding the mapping $G \to p(G)$, which will save us $|G| \log(N)$ bits, however we have to pay an extra $|G| \log(eN/|G|)$ bits to encode the set $p(G)$, so overall we compressed by $|G| \log(|G|/e)$ bits, and therefore $S \geq |G|$ assuming $|G| \geq 2e$. Thus the question is how large a set G can we choose. A simple probabilistic argument, basically picking values at random until it's no longer possible to extend G, shows that we can always pick a G of size at least $|G| \geq \epsilon N/T$, and we conclude $S \geq \epsilon N/T$ assuming $T \leq \epsilon N/2e$.

In the De et al. proof, the size of the good set G will always be close to $\epsilon N/T$, no matter how $\mathcal{A}_{\mathsf{aux}}$ actually behaves. In this paper we give a more fine grained analysis introducing a new parameter T_g as discussed next.

The T_g parameter. Informally, our compression algorithm for a function $g : [N] \to [N]$ goes as follows: Define the set $I = \{x \ : \ \mathcal{A}^g_{\mathsf{aux}}(g(x)) = x\}$ of values where $\mathcal{A}^g_{\mathsf{aux}}$ inverts $g(I)$, by assumption $|I| = \epsilon N$. Now we can add values from I to G as long as possible, every time we add a value x, we "spoil" up to T values in I, where we say x' gets spoiled if $\mathcal{A}^g_{\mathsf{aux}}(g(x))$ makes oracle query x', and thus we will not be able to add x' to G in the future. As we start with $|I| = \epsilon N$, and spoil at most T values for every value added to G, we can add at least $\epsilon N/T$ values to G.

This is a worst case analysis assuming $\mathcal{A}^g_{\mathsf{aux}}$ really spoils close to T values every time we add a value to G, but potentially $\mathcal{A}^g_{\mathsf{aux}}$ behaves nicer and on average spoils less. In the proof of Lemma 1 we take advantage of this and extend G as

long as possible, ending up with a good set G of size at least $\epsilon N/2T_g$ for some $1 \leq T_g \leq T$. Here T_g is the average number of elements we spoiled for every element added to G.

This doesn't help to improve the De et al. lower bound, as in general T_g can be as large as T in which case our lower bound $S \cdot T_g \in \Omega(\epsilon N)$ coincides with the De at al. $S \cdot T \in \Omega(\epsilon N)$ lower bound.[18] But this more fine grained bound will be a crucial tool to prove the lower bound for g_f.

Lower Bound for g_f. We now outline the proof idea for our lower bound $S^2 \cdot T \in \Omega(\epsilon^2 N^2)$ for inverting $g_f(x) = g(x, x'), f(x) = \overline{f(x')}$ assuming $g : [N] \times [N] \to [N]$ is a random function and $f : [N] \to [N]$ is a random permutation. We assume an adversary $\mathcal{A}^{g,f}_{\mathsf{aux}}$ exists which has oracle access to f, g and inverts $g_f : [N] \to [N]$ on a set $J = \{y \ : \ g_f(\mathcal{A}^{f,g}_{\mathsf{aux}}(y)) = y\}$ of size $J = |\epsilon N|$.

If the function table of f is given, $g_f : [N] \to [N]$ is a random function that can be efficiently evaluated, and we can prove a lower bound $S \cdot T_g \in \Omega(\epsilon N)$ as outlined above.

At this point, we make a case distinction, depending on whether T_g is below or above $\sqrt{T}$.

If $T_g < \sqrt{T}$ our $S \cdot T_g \in \Omega(\epsilon N)$ bound becomes $S^2 \cdot T \in \Omega(\epsilon^2 N^2)$ and we are done.

The more complicated case is when $T_g \geq \sqrt{T}$ where we show how to use the existence of $\mathcal{A}^{f,g}_{\mathsf{aux}}$ to compress f instead of g. Recall that T_g is the average number of values that got "spoiled" while running the compression algorithm for g_f, that means, for every value added to the good set G, $\mathcal{A}^{f,g}_{\mathsf{aux}}$ made on average T_g "fresh" queries to g_f. Now making fresh g_f queries isn't that easy, as it requires finding x, x' where $f(x) = \overline{f(x')}$. We can use $\mathcal{A}^{f,g}_{\mathsf{aux}}$ which makes many such fresh g_f queries to "compress" f: when $\mathcal{A}^{f,g}_{\mathsf{aux}}$ makes two f queries x, x' where $f(x) = \overline{f(x')}$, we just need to store the first output $f(x)$, but won't need the second $f(x')$ as we know it is $\overline{f(x)}$. For decoding we also must store when exactly $\mathcal{A}^{f,g}_{\mathsf{aux}}$ makes the f queries x and x', more on this below.

Every time we invoke $\mathcal{A}^{f,g}_{\mathsf{aux}}$ for compression as just outlined, up to T outputs of f may get "spoiled" in the sense that $\mathcal{A}^{f,g}_{\mathsf{aux}}$ makes an f query that we need to answer at this point, and thus it's no longer available to be compressed later.

As $\mathcal{A}^{f,g}_{\mathsf{aux}}$ can spoil up to T queries on every invocation, we can hope to invoke it at least $\epsilon N/T$ times before all the f queries are spoiled. Moreover, on average $\mathcal{A}^{f,g}_{\mathsf{aux}}$ makes T_g fresh g_f queries, so we can hope to compress around T_g outputs of f with every invocation of $\mathcal{A}^{f,g}_{\mathsf{aux}}$, which would give us around $T_g \cdot \epsilon N/T$ compressed values. This assumes that a large fraction of the fresh g_f queries uses values of f that were not spoiled in previous invocations. The technical core of our proof is a combinatorial lemma which we state and prove in §5, which implies that it's always possible to find a sequence of inputs to $\mathcal{A}^{f,g}_{\mathsf{aux}}$ such that this is the case.

[18] Note that for the adversary as specified by Hellman's attack against permutations as outlined in §1.4 we do have $T_g = T$, which is not surprising given that for permutations the De at al. lower bound matches Hellman's attack.

Concretely, we can always find a sequence of inputs such that at least $T_g \cdot \epsilon N/32T$ values can be compressed.[19]

2 Notation and Basic Facts

We use brackets like $(x_1, x_2, \ldots)$ and $\{x_1, x_2, \ldots\}$ to denote ordered and unordered sets, respectively. We'll usually refer to unordered sets simply as sets, and to ordered sets as lists. $[N]$ denotes some domain of size N, for notational convenience we assume $N = 2^n$ is a power of two and identify $[N]$ with $\{0,1\}^n$. For a function $f : [N] \to [M]$ and a set $S \subseteq [N]$ we denote with $f(S)$ the set $\{f(S[1]), \ldots, f(S[|S|])\}$, similarly for a list $L \subseteq [N]$ we denote with $f(L)$ the list $(f(L[1]), \ldots, f(L[|L|]))$. For a set $\mathcal{X}$, we denote with $x \leftarrow \mathcal{X}$ that x is assigned a value chosen uniformly at random from $\mathcal{X}$.

Fact 1 (from [DTT10]) *For any randomized encoding procedure* $\mathsf{Enc} : \{0,1\}^r \times \{0,1\}^n \to \{0,1\}^m$ *and decoding procedure* $\mathsf{Dec} : \{0,1\}^r \times \{0,1\}^m \to \{0,1\}^n$ *where*

$$\Pr_{x \leftarrow \{0,1\}^n, \rho \leftarrow \{0,1\}^r}[\mathsf{Dec}(\rho, \mathsf{Enc}(\rho, x)) = x] \geq \delta$$

we have $m \geq n - \log(1/\delta)$.

Fact 2 *If a set* X *is at least* ϵ *dense in* Y*, i.e.,* $X \subset Y$, $|X| \geq \epsilon |Y|$*, and* Y *is known, then* X *can be encoded using* $|X| \cdot \log(e/\epsilon)$ *bits. To show this we use the inequality* $\binom{n}{\epsilon n} \leq (en/\epsilon n)^{\epsilon n}$*, which implies* $\log \binom{n}{\epsilon n} \leq \epsilon n \log(e/\epsilon)$.

3 A Lower Bound for Functions

The following theorem is basically from [DTT10], but stated for functions not permutations.

Theorem 1. *Fix some* $\epsilon \geq 0$ *and an oracle algorithm* $\mathcal{A}$ *which on any input makes at most* T *oracle queries. If for every function* $f : [N] \to [N]$ *there exists a string* aux *of length* $|\mathsf{aux}| = S$ *such that*

$$\Pr_{y \leftarrow [N]}[f(\mathcal{A}^f_{\mathsf{aux}}(y)) = y] \geq \epsilon$$

then

$$T \cdot S \in \Omega(\epsilon N) \ . \tag{4}$$

The theorem follows from Lemma 1 below using Fact 1 as follows: in Fact 1 let $\delta = 0.9$ and $n = N \log N$, think of x as the function table of a function $f : [N] \to [N]$. Then $|\mathsf{Enc}(\rho, \mathsf{aux}, f)| \geq N \log N - \log(1/0.9)$, together with the upper bound on the encoding from Eq.(6) this implies Eq.(4). Note that the extra assumption that $T \leq \epsilon N/40$ in the lemma below doesn't matter, as if it's not satisfied the theorem is trivially true. For now the value T_g in the lemma below is not important and the reader can just assume $T_g = T$.

[19] The constant 32 here can be decreased with a more fine-grained analysis, we opted for a simpler proof rather than optimising this constant.

Lemma 1. *Let $\mathcal{A}, T, S, \epsilon$ and f be as in Theorem 1, and assume $T \leq \epsilon N/40$. There are randomized encoding and decoding procedures $\mathsf{Enc}, \mathsf{Dec}$ such that if $f : [N] \to [N]$ is a function and for some aux of length $|\mathsf{aux}| = S$*

$$\Pr_{y \leftarrow [N]}[f(\mathcal{A}^f_{\mathsf{aux}}(y)) = y] \geq \epsilon$$

then

$$\Pr_{\rho \leftarrow \{0,1\}^r}[\mathsf{Dec}(\rho, \mathsf{Enc}(\rho, \mathsf{aux}, f)) = f] \geq 0.9 \tag{5}$$

and the length of the encoding is at most

$$|\mathsf{Enc}(\rho, \mathsf{aux}, f)| \leq \underbrace{N \log N}_{=|f|} - \frac{\epsilon N}{2T_g} + S + \log(N) \tag{6}$$

for some $T_g, 1 \leq T_g \leq T$.

3.1 Proof of Lemma 1

The Encoding and Decoding Algorithms. In Algorithms 1 and 2, we always assume that if $\mathcal{A}^f_{\mathsf{aux}}(y)$ outputs some value x, it makes the query $f(x)$ at some point. This is basically w.l.o.g. as we can turn any adversary into one satisfying this by making at most one extra query. If at some point $\mathcal{A}^f_{\mathsf{aux}}(y)$ makes an oracle query x where $f(x) = y$, then we also w.l.o.g. assume that right after this query $\mathcal{A}$ outputs x and stops. Note that if $\mathcal{A}$ is probabilistic, it uses random coins which are given as input to $\mathsf{Enc}, \mathsf{Dec}$, so we can make sure the same coins are used during encoding and decoding.

The Size of the Encoding. We will now upper bound the size of the encoding of $G, f(Q'), (|q_1|, \ldots, |q_{|G|}|), f([N] - \{G^{-1} \cup Q'\})$ as output in line (15) of the Enc algorithm.

Let $T_g := |B|/|G|$ be the average number of elements we added to the bad set B for every element added to the good set G, then

$$|G| \geq \epsilon N/2T_g \ . \tag{7}$$

To see this we note that when we leave the while loop (see line (8) of the algorithm Enc) it holds that $|B| \geq |J|/2 = \epsilon N/2$, so $|G| = |B|/T_g \geq |J|/2T_g = \epsilon N/2T_g$.

G: Instead of G we will actually encode the set $\pi^{-1}(G) = \{c_1, \ldots, c_{|G|}\}$. From this the decoding Dec (who gets ρ, and thus knows π) can then reconstruct $G = \pi(\pi^{-1}(G))$. We claim that the elements in $c_1 < c_2 < \ldots < c_{|G|}$ are whp. at least $\epsilon/2$ dense in $[c_{|G|}]$ (equivalently, $c_{|G|} \leq 2|G|/\epsilon$). By Fact 2 we can thus encode $\pi^{-1}(G)$ using $|G| \log(2e/\epsilon) + \log N$ bits (the extra $\log N$ bits are used to encode the size of G which is required so decoding later knows how to parse the encoding). To see that the c_i's are $\epsilon/2$ dense whp. consider line (9) in Enc which states $c := \min\{c' > c \ : \ y_{c'} \in \{J \setminus B\}\}$. If we replace $J \setminus B$

Algorithm 1 Enc

Input: $\mathcal{A}, \mathsf{aux}$, randomness ρ and a function $f : [N] \to [N]$ to compress.
Initialize: $B, G := \emptyset, c := -1$
Throughout we identify $[N]$ with $\{0, \ldots, N-1\}$.
Pick a random permutation $\pi : [N] \to [N]$ (using random coins from ρ)
Let $J := \{y \ : \ f(\mathcal{A}^f_{\mathsf{aux}}(y)) = y\}$, $|J| = \epsilon N$ ▷ The set J where $\mathcal{A}$ inverts. If $\mathcal{A}$ is probabilistic, use random coins from ρ.
For $i = 0, \ldots, N-1$ define $y_i := \pi(i)$. ▷ Randomize the order
For $y \in J$ let the list $q(y)$ contain all queries made by $A^f(y)$ except the last query (which is x s.t. $f(x) = y$).
while $|B| < |J|/2$ **do** ▷ While the bad set contains less than half of J
    $c := \min\{c' > c \ : \ y_{c'} \in \{J \setminus B\}\}$ ▷ Increase c to the next y_c in $J \setminus B$
    $G := G \cup y_c$ ▷ Add this y_c to good set
    $B := B \cup (f(q(y_c)) \cap J)$ ▷ Add spoiled queries to bad set
end while
Let $G = \{g_1, \ldots, g_{|G|}\}, Q = (q(g_1), \ldots, q(g_{|G|}))$, and define $Q' = (q'_1, \ldots, q'_{|G|}), q'_i \subseteq q(g_i)$ to contain only the "fresh" queries in Q by deleting all but the first occurrence of every element. E.g. if $(q(g_1), q(g_2)) = ((1,2,3,1),(2,4,5,4))$ then $(q'_1, q'_2) = ((1,2,3),(4,5))$.
Let $G^{-1} = \{\mathcal{A}^f_{\mathsf{aux}}(y) \ : \ y \in G\}$
Output an encoding of (the set) G, (the lists) $f(Q'), (|q'_1|, \ldots, |q'_{|G|}|), f([N] - \{G^{-1} \cup Q'\})$ and (the string) aux.

with J, then the c_i's would be whp. close to ϵ dense as J is ϵ dense in $[N]$ and the y_i are uniformly random. As $|B| < |J|/2$, using $J \setminus B$ instead of J will decrease the density by at most a factor 2. If we don't have this density, i.e., $c_{|G|} > 2|G|/\epsilon$, we consider encoding to have failed.

$f(Q')$: This is a list of Q' elements in $[N]$ and can be encoded using $|Q'| \log N$ bits.

$(|q'_1|, \ldots, |q'_{|G|}|)$: Require $|G| \log T$ bits as $|q'_i| \leq |q_i| \leq T$. A more careful argument (using that the q'_i are on average at most T_g) requires $|G| \log(eT_g)$ bits.

$f([N] - \{G^{-1} \cup Q'\})$: Requires $(N - |G| - |Q'|) \log N$ bits (using that $G^{-1} \cap Q' = \emptyset$ and $|G^{-1}| = |G|$).

aux: Is S bits long.

Summing up we get

$$|\mathsf{Enc}(\rho, \mathsf{aux}, f)| = |G| \log(2e^2 T_g/\epsilon) + (N - |G|) \log N + S + \log N$$

as by assumption $T_g \leq T \leq \epsilon N/40$, we get $\log N - \log(2e^2 T_g/\epsilon) \geq 1$, and further using (7) we get

$$|\mathsf{Enc}(\rho, \mathsf{aux}, f)| \leq N \log N - \frac{\epsilon N}{2T_g} + S + \log N$$

as claimed.

Algorithm 2 Dec

1: **Input:** $\mathcal{A}, \rho$ and the encoding $(G, f(Q'), (|q'_1|, \ldots, |q'_{|G|}|), f([N]-\{G^{-1} \cup Q'\}), \mathsf{aux})$.
2: Let π be as in Enc.
3: Let $(g_1, \ldots, g_{|G|})$ be the elements of G ordered as they were added by Enc (i.e., $\pi^{-1}(g_i) < \pi^{-1}(g_{i+1})$ for all i).
4: Invoke $\mathcal{A}^{(\cdot)}_{\mathsf{aux}}(\cdot)$ sequentially on inputs $g_1, \ldots, g_{|G|}$ using $f(Q')$ to answer $\mathcal{A}_{\mathsf{aux}}$'s oracle queries. ▷ If $\mathcal{A}$ is probabilistic, use the same random coins from ρ as in Enc.
5: Combine the mapping $G^{-1} \cup Q' \to f(G^{-1} \cup Q')$ (which we learned in the previous step) with $[N]-\{G^{-1} \cup Q'\} \to f([N]-\{G^{-1} \cup Q'\})$ to learn the entire $[N] \to f([N])$
6: Output $f([N])$

4 A Lower Bound for $g(x, f^{-1}(\overline{f(x)}))$

For a permutation $f : [N] \to [N]$ and a function $g : [N] \times [N] \to [N]$ we define $g_f : [N] \to [N]$ as

$$g_f(x) = g(x, x') \text{ where } f(x) = \overline{f(x')} \text{ or equivalently } g_f(x) = g(x, f^{-1}(\overline{f(x)}))$$

Theorem 2. *Fix some $\epsilon > 0$ and an oracle algorithm $\mathcal{A}$ which makes at most*

$$T \leq (N/4e)^{2/3} \tag{8}$$

oracle queries and takes an advice string aux of length $|\mathsf{aux}| = S$. If for all functions $f : [N] \to [N], g : [N] \times [N] \to [N]$ and some aux of length $|\mathsf{aux}| = S$ we have

$$\Pr_{y \leftarrow [N]}[g_f(A^{f,g}_{\mathsf{aux}}(y)) = y] \geq \epsilon \tag{9}$$

then

$$TS^2 \in \Omega(\epsilon^2 N^2) \ . \tag{10}$$

The theorem follows from Lemma 2 below as we'll prove thereafter.

Lemma 2. *Fix some $\epsilon \geq 0$ and an oracle algorithm $\mathcal{A}$ which makes at most $T \leq (N/4e)^{2/3}$ oracle queries. There are randomized encoding and decoding procedures $\mathsf{Enc}_g, \mathsf{Dec}_g$ and $\mathsf{Enc}_f, \mathsf{Dec}_f$ such that if $f : [N] \to [N]$ is a permutation, $g : [N] \times [N] \to [N]$ is a function and for some advice string aux of length $|\mathsf{aux}| = S$ we have*

$$\Pr_{y \leftarrow [N]}[g_f(A^{f,g}_{\mathsf{aux}}(y)) = y] \geq \epsilon$$

then

$$\Pr_{\rho \leftarrow \{0,1\}^r}[\mathsf{Dec}_g(\rho, f, \mathsf{Enc}_g(\rho, \mathsf{aux}, f, g)) = g] \geq 0.9 \tag{11}$$

$$\Pr_{\rho \leftarrow \{0,1\}^r}[\mathsf{Dec}_f(\rho, g, \mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)) = f] \geq 0.9 \ . \tag{12}$$

Moreover for every ρ, aux, f, g there is a $T_g, 1 \leq T_g \leq T$, such that

$$|\mathsf{Enc}_g(\rho, \mathsf{aux}, f, g)| \leq \underbrace{N^2 \log N}_{=|g|} - \frac{\epsilon N}{2T_g} + S + \log N \tag{13}$$

and if $T_g \geq \sqrt{T}$

$$|\mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)| \leq \underbrace{\log N!}_{=|f|} - \frac{\epsilon N T_g}{64T} + S + \log N \ . \tag{14}$$

We first explain how Theorem 2 follows from Lemma 2 using Fact 1.

Proof (of Theorem 2). The basic idea is to make a case analysis; if $T_g < \sqrt{T}$ we compress g, otherwise we compress f. Intuitively, our encoding for g achieving Eq.(13) makes both f and g queries, but only g queries "spoil" g values. As the compression runs until all g values are spoiled, it compresses better the smaller T_g is. On the other hand, the encoding for f achieving Eq.(12) is derived from our encoding for g, and it manages to compresses in the order of T_g values of f for every invocation (while "spoiling" at most T of the f values), so the larger T_g the better it compresses f.

Concretely, pick f, g uniformly at random (and assume Eq.(9) holds). By a union bound for at least a 0.8 fraction of the ρ Eq.(11) and Eq.(12) hold simultaneously. Consider any such good ρ, which together with f, g fixes some $T_g, 1 \leq T_g \leq T$ as in the statement of Lemma 2. Now consider an encoding $\mathsf{Enc}_{f,g}$ where $\mathsf{Enc}_{f,g}(\rho, \mathsf{aux}, f, g)$ outputs $(f, \mathsf{Enc}_g(\rho, \mathsf{aux}, f, g))$ if $T_g < \sqrt{T}$, and $(g, \mathsf{Enc}_f(\rho, \mathsf{aux}, f, g))$ otherwise.

- If $T_g < \sqrt{T}$ we use (13) to get

  $$|\mathsf{Enc}_{f,g}(\rho, \mathsf{aux}, f, g)| = |f| + |\mathsf{Enc}_g(\rho, \mathsf{aux}, f, g)| \leq |f| + |g| - \epsilon N/2T_g + S + \log N$$

  and now using Fact 1 (with $\delta = 0.8$) we get

  $$S \geq \epsilon N/2T_g - \log N - \log(1/0.8) > \epsilon N/2\sqrt{T} - \log N - \log(1/0.8)$$

  and thus $TS^2 \in \Omega(\epsilon^2 N^2)$ as claimed in Eq.(10).
- If $T_g \geq \sqrt{T}$ then we use Eq.(14) and Fact 1 and again get $S \geq \epsilon N T_g/64T - \log N - \log(1/0.8)$ which implies Eq.(10) as $T_g \geq \sqrt{T}$.

□

Algorithm 3 Enc_g

1: **Input:** $\mathcal{A}, \rho, \mathsf{aux}, f, g$
2: Compute the function table of $g_f : [N] \to [N]$, $g_f(x) = g(x, x')$ where $f(x) = \overline{f(x')}$.
3: Invoke $E_{g_f} \leftarrow \mathsf{Enc}(\mathcal{A}, g_f, aux, \rho)$
4: Let g' be the function table of $g([N]^2) = g(1,1) \| \ldots \| g(N,N)$, but with the N entries (x, x') where $f(x) = \overline{f(x')}$ deleted.
5: Output $E_{g_f}, g', \mathsf{aux}$.

Algorithm 4 Dec_g

Input: $\mathcal{A}, \rho, f$ and the encoding $(E_{g_f}, g', \mathsf{aux})$ of g.
Invoke $g_f \leftarrow \mathsf{Dec}(\mathcal{A}, \rho, \mathsf{aux}, E_{g_f})$.
Reconstruct g from g' and g_f (this is possible as f is given).
Output $g([N]^2)$

Algorithm 5 Enc_f

Input: $\mathcal{A}, \rho, \mathsf{aux}, f, g$
Invoke $E_{g_f} \leftarrow \mathsf{Enc}(\mathcal{A}, g_f, \mathsf{aux}, \rho)$ ▷ Compute the same encoding of g_f as Enc_g did.
For $G \in E_{g_f}$, let $G_f \subset G, G_f = \{z_1, \ldots, z_{|G_f|}\}$ be as defined in proof of Lemma 2.
Initialize empty lists $L_f, T_f, C_f := \emptyset$.
for $i = 1$ to $|G_f|$ **do**
    Invoke $\mathcal{A}^{f,g}_{\mathsf{aux}}(z_i)$. ▷ Using random coins from ρ if $\mathcal{A}$ is probabilistic.
    For each pair of f queries x, x' (made in this order during invocation) where $f(x) = \overline{f(x')}$ and neither $f(x')$ nor $f(x)$ is in $L_f \cup C_f$, let (t, t') be the indices $(1 \leq t < t' \leq T)$ specifying when during invocation these queries were made. Append (t, t') to T_f and append $f(x')$ to C_f.
    Append all images of oracle queries to f made during invocation of $\mathcal{A}^{f,g}_{\mathsf{aux}}(z_i)$ to L_f, except if the value is in $L_f \cup C_f$. ▷ Append the images of all fresh f queries which were not compressed.
end for
Let L_f^{-1} (similarly C_f^{-1}) contain the inputs corresponding to L_f, i.e., add x to L_f^{-1} when adding $f(x)$ to L_f.
Output an encoding of G_f, the list of values of f queries L_f, the list of tuples T_f and the remaining outputs $f([N - L_f^{-1} - C_f^{-1}])$ which were neither in the list L_f nor compressed.

Algorithm 6 Dec_f

Input: $\mathcal{A}, \rho, \mathsf{aux}, g$ and encoding $(G_f, L_f, T_f, f([N - L_f^{-1} - C_f^{-1}]))$ of f.
Let $G_f = \{z_1, \ldots, z_{|G_f|}\}$.
for $i = 1$ to $|G_f|$ **do**
    Invoke $\mathcal{A}^{(\cdot),g}_{\mathsf{aux}}(z_i)$ reconstructing the answers to the first oracle (which should be f) using the lists L_f and T_f.
end for
For L_f^{-1}, C_f^{-1} as in Enc_f, we have learned the mapping $(L_f^{-1} \cup C_f^{-1}) \to f((L_f^{-1} \cup C_f^{-1}))$. Reconstruct all of $f([N])$ by combining this with $f([N] - (L_f^{-1} \cup C_f^{-1}))$.
Output $f([N])$

Proof (of Lemma 2).

The Encoding and Decoding Algorithms. The encoding and decoding of g are depicted in Algorithms 3 and 4, and those of f in Algorithms 5 and 6. $\mathcal{A}^{f,g}_{\mathsf{aux}}(\cdot)$ can make up to T queries in total to its oracles $f(.)$ and $g(.)$. We will assume that whenever a query $g(x, x')$ is made, the adversary made queries $f(x)$ and $f(x')$ before. This is basically without loss of generality as we can turn any adversary into one adhering to this by at most tripling the number of queries. It will also be convenient to assume that $\mathcal{A}^{f,g}_{\mathsf{aux}}$ only queries g on its restriction to g_f, that is, for all $g(x, x')$ queries it holds that $f(x) = \overline{f(x')}$, but the proof is easily extended to allow all queries to g as our encoding will store the function table of g on all "uninteresting" inputs $(x, x'), f(x) \neq \overline{f(x')}$ and thus can directly answer any such query.

As in the proof of Lemma 1, we don't explicitly show the randomness in case $\mathcal{A}$ is probabilistic.

The Size of the Encodings. We will now upper bound the size of the encodings output by Enc_g and Enc_f in Algorithms 3 and 5 and hence prove Eq.(13) and Eq.(14).

Eq.(13) now follows almost directly from Theorem 1 as our compression algorithm Enc_g for $g : [N] \times [N] \to [N]$ simply uses Enc to compress g restricted to $g_f : [N] \to [N]$, and thus compresses by exactly the same amount as Enc.

It remains to prove an upper bound on the length of the encoding of f by our algorithm Enc_f as claimed in Eq.(14). Recall that Enc (as used inside Enc_g) defines a set G such that for every $y \in G$ we have (1) $A^{f,g}_{\mathsf{aux}}(y)$ inverts, i.e., $g_f(A^{f,g}_{\mathsf{aux}}(y)) = y$ and (2) never makes a g_f query x where $g_f(x) \in G$. Recall that T_g in Eq.(13) satisfies $T_g = \epsilon N/2|G|$, and corresponds to the average number of "fresh" g_f queries made by $A^{f,g}_{\mathsf{aux}}(\cdot)$ when invoked on the values in G.

Enc_f invokes $A^{f,g}_{\mathsf{aux}}(\cdot)$ on a carefully chosen subset $G_f = (z_1, \ldots, z_{|G_f|})$ of G (to be defined later). It keeps lists L_f, C_f and T_f such that after invoking $A^{f,g}_{\mathsf{aux}}(\cdot)$ on G_f, $L_f \cup C_f$ holds the outputs to all f queries made. Looking ahead, the decoding Dec_f will also invoke $A^{f,g}_{\mathsf{aux}}(\cdot)$ on G_f, but will only need L_f and T_f (but not C_f) to answer all f queries.

The lists L_f, T_f, C_f are generated as follows. On the first invocation $A^{f,g}_{\mathsf{aux}}(z_1)$ we observe up to T oracle queries made to g and f. Every g query (x, x') must be preceded by f queries x and x' where $f(x) = \overline{f(x')}$. Assume x and x' are the queries number t, t' $(1 \leq t < t' \leq T)$. A key observation is that by just storing (t, t') and $f(x)$, Dec_f will later be able to reconstruct $f(x')$ by invoking $A^{f,g}_{\mathsf{aux}}(z_1)$, and when query t' is made, looking up the query $f(x)$ in L_f (its position in L_f is given by t), and set $f(x') = \overline{f(x)}$. Thus, every time a fresh query $f(x')$ is made we append it to L_f, unless earlier in this invocation we made a fresh query $f(x)$ where $f(x') = \overline{f(x)}$. In this case we append the indices (t, t') to the list T_f. We also add $f(x')$ to a list C_f just to keep track of what we already compressed. Enc_f now continues this process by invoking $A^{f,g}_{\mathsf{aux}}(\cdot)$ on inputs $z_2, z_3, \ldots, z_{|G_f|} \in G_f$ and finally outputs and encoding of G_f, an encoding of the list of images of fresh queries L_f, an encoding of the list of colliding indices T_f, aux, and all values of f that were neither compressed nor queried.

In the sequel we show how to choose $G_f \in G$ such that $|G_f| \geq \epsilon N/8T$ and hence it can be encoded using $|G_f| \log N + \log N$ where the extra $\log N$ is used to encode $|G_f|$. We also show that $|T_f| \geq |G_f| \cdot T_g/4$ and furthermore that we can compress at least one bit per element of T_f. Putting things together we get

$$|\mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)| \leq \log N! - |G_f|(T_g/4 - \log N) + S + \log N \ .$$

And if $\log N \leq T_g/8$, we get Eq.(14)

$$|\mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)| \leq \log N! - \epsilon N T_g/64T + S + \log N \ .$$

Given G such that $|G| \geq \epsilon N/2T_g$, the subset G_f can be constructed by carefully applying Lemma 3 which we prove in Sec. 5. Let $(X_1, \ldots, X_{|G|})$, $(Y_1, \ldots, Y_{|G|})$ be two sequences of sets such that $Y_i \subseteq X_i \subseteq [N]$ and $|X_i| \leq T$ such that Y_i and X_i respectively correspond to g and f queries in $|G|$ consecutive executions of $A^{f,g}_{\mathsf{aux}}(\cdot)$ on G.[20] Given such sequences Lemma 3 constructs a subsequence of executions $G_f \subseteq G$ whose corresponding g queries $(Y_{i_1}, \ldots, Y_{i_{|G_f|}})$ are fresh. As a g query is preceded by two f queries, such a subsequence induces a sequence $(Z_{i_1}, \ldots, Z_{i_{|G_f|}})$ of queries that are not only fresh for g but also fresh for f. Furthermore, such a sequence covers $y \cdot |I|/16T$ where $y = |I|/|G|$ is the average coverage of Y_i's and $I \subseteq [N]$ is their total coverage.

However, Lemma 3 considers a g query $(x, x') \in Y_i$ to be fresh if either $x \notin \cup_{j=1}^{i-1} X_j$ or $x' \notin \cup_{j=1}^{i-1} X_j$, i.e., if at least one of x, x' is fresh in the i^{th} execution, then the pair is considered fresh. For compressing f both x, x' need to be fresh. To enforce that and apply Lemma 3 directly, we apply Lemma 3 on augmented sets $X_1, \ldots, X_{|G|}$ such that whenever X_i, Y_i are selected, the corresponding Z_i contains exactly $|Z_i|/2$ pairs of queries that are fresh for both g and f. We augment X_i as follows. For every X_i and every f query x made in the i^{th} step, add $f^{-1}(\overline{f(x)})$ to X_i. This augmentation results in X_i such that $|X_i| \leq 2T$ as originally we have $|X_i| \leq T$.

Applying Lemma 3 on $Y_1, \ldots, Y_{|G|}$ and such augmented sets $X_1, \ldots, X_{|G|}$ yields G_f such that the total number of fresh colliding *queries* is of size at least

$$y \cdot \frac{|I|}{16 \cdot 2T} = \frac{\epsilon N}{|G|} \cdot \frac{\epsilon N}{32T} = \frac{\epsilon N T_g}{16T} \ .$$

Therefore the total number of fresh colliding *pairs*, or equivalently $|T_f|$, is $\epsilon N T_g/32T$ as claimed. Furthermore, Lemma 3 guarantees that $|G_f| \geq \epsilon N/8T$.[21]

What remains to show is that for each colliding pair in T_f we compress by at least one bit. Recall that the list T_f has exactly as many entries as C_f. However

[20] Here is how these sets are compiled. Note that if q is an f query then $q \in [N]$, and if q is a g query then $q \in [N]^2$. In the i^{th} execution, both X_i, Y_i are initially empty and later will contain only elements in $[N]$. Therefore for each query q, if $q = (x, x')$ is a g query we add two elements x and x' to Y_i, and if $q = x$ is an f query we add the single element x to X_i. Furthermore as a g query (x, x') is preceded by two f queries x, x', then $Y_i \subseteq X_i$, and as the max number of queries is T we have $|X_i| \leq T$.

[21] $|G_f|$ corresponds to ℓ in the proof of Lemma 3.

entries in T_f are colliding pairs of indices (t, t') and entries in C_f are images of size $\log N$. Per each entry (t, t') in T_f we compress if the encoding size of (t, t') is strictly less than $\log N$. Here is an encoding of T_f that achieves this. Instead of encoding each entry (t, t') as two indices which costs $2 \log T$ and therefore we save one bit per element in T_f assuming $T \leq \sqrt{N/2}$, we encode the set of colliding pairs among all possible query pairs. Concretely, for each $z \in G_f$ we obtain a set of colliding indices of size at least $T_g/4$. Then we encode this set of colliding pairs $T_g/4$ among all possible pairs[22], which is upper bounded by T^2, using

$$\log \binom{T^2}{T_g/4} \leq \frac{T_g}{4} \log \frac{4eT^2}{T_g}$$

bits, and therefore, given that $T_g \geq \sqrt{T}$ and $T \leq (N/4e)^{2/3}$, we have that $\log N - \log 4eT^2/T_g \geq 1$ and therefore we compress by at least one bit for each pair, i.e., for each element in T_f, and that concludes the proof.

□

5 A Combinatorial Lemma

In this section we state and prove a lemma which can be cast in terms of the following game between Alice and Bob. For some integers n, N, M, Alice can choose a partition $(Y_1, \ldots, Y_n)$ of $I \subseteq [N]$, and for every Y_i also a superset $X_i \supseteq Y_i$ of size $|X_i| \leq M$. The goal of Bob is to find a subsequence $1 \leq b_1 < b_2 < \ldots < b_\ell$ such that $Y_{b_1}, Y_{b_2}, \ldots, Y_{b_\ell}$ contains as many "fresh" elements as possible, where after picking Y_{b_i} the elements $\bigcup_{k=1}^{i} X_{b_k}$ are not fresh, i.e., picking Y_{b_i} "spoils" all of X_{b_i}. How many fresh elements can Bob expect to hit in the worst case? Intuitively, as every Y_{b_i} added spoils up to M elements, he can hope to pick up to $\ell \approx |I|/M$ of the Y_i's before most of the elements are spoiled. As the Y_i are on average of size $y := |I|/n$, this is also an upper bound on the number of fresh elements he can hope to get with every step. This gives something in the order of $y \cdot (|I|/M)$ fresh elements in total. By the lemma below a subsequence that contains about that many fresh elements always exists.

Lemma 3. *For $M, N \in \mathbb{N}, M \leq N$ and any disjoint sets $Y_1, \ldots, Y_n \subset [N]$*

$$\bigcup_{i=1}^{n} Y_i = I \quad , \quad \forall i \neq j : Y_i \cap Y_j = \emptyset$$

and supersets $(X_1, \ldots, X_n)$ where

$$\forall i \in [n] \ : \ Y_i \subseteq X_i \subseteq [N] \quad , \quad |X_i| \leq M$$

[22] Note that T^2 is an upper bound on all possible pairs of queries, however as we have that $t < t'$ for each pair (t, t'), we can cut T^2 by at least a factor of 2. Other optimizations are possible. This extra saving one can use to add extra dummy pairs of indices to separate executions for decoding. The details are tedious and do not affect the bound as we were generous to consider T^2 to be the size of possible pairs.

there exists a subsequence $1 \leq b_1 < b_2 < \ldots < b_\ell \leq n$ *such that the sets*

$$Z_{b_j} = Y_{b_j} \setminus \cup_{k<j} X_{b_k} \tag{15}$$

have total size

$$\sum_{j=1}^{\ell} |Z_{b_j}| = |\bigcup_{j=1}^{\ell} Z_{b_j}| \geq y \cdot \frac{|I|}{16M}$$

where $y = |I|/n$ *denotes the average size of the* Y_i*'s.*

Proof. Let $(Y_{a_1}, \ldots, Y_{a_m})$ be a subquence of $(Y_1, \ldots, Y_n)$ that contains all the sets of size at least $y/2$. By a Markov bound, these large Y_{a_i}'s cover at least half of the domain I, i.e.

$$\left|\cup_{i\in[m]} Y_{a_i}\right| > |I|/2 \ . \tag{16}$$

We now choose the subsequence $(Y_{b_1}, \ldots, Y_{b_\ell})$ from the statement of the lemma as a subsequence of $(Y_{a_1}, \ldots, Y_{a_m})$ in a greedy way: for $i = 1, \ldots, m$ we add Y_{a_i} to the sequence if it adds a lot of "fresh" elements, concretely, assume we are in step i and so far have added $Y_{b_1}, \ldots, Y_{b_{j-1}}$, then we'll pick the next element, i.e., $Y_{b_j} := Y_{a_i}$, if the fresh elements $Z_{b_j} = Y_{b_j} \setminus \cup_{k<j} X_{b_k}$ contributed by Y_{b_j} are of size at least $|Z_{b_j}| > |Y_{b_j}|/2$.

We claim that we can always add at least one more Y_{b_j} as long as we haven't yet added at least $|I|/4M$ sets, i.e., $j < |I|/4M$. Note that this then proves the lemma as

$$\sum_{j=1}^{\ell} |Z_{b_j}| \geq \sum_{j=1}^{\ell} |Y_{b_j}|/2 \geq \ell y/4 \geq |I|/4M \cdot y/4 = y|I|/16M \ .$$

It remains to prove the claim. For contradiction assume our greedy algorithm picked $(Y_{b_1}, \ldots, Y_{b_\ell})$ with $\ell < |I|/4M$. We'll show that there is a Y_{a_t} (with $a_t > b_\ell$) with

$$|Y_{a_t} \setminus \cup_{i=1}^{j} X_{b_i}| \geq |Y_{a_t}|/2$$

which is a contradiction as this means the sequence could be extended by $Y_{b_{\ell+1}} = Y_{a_t}$. We have

$$|\cup_{i=1}^{\ell} X_{b_i}| \leq |I|/4M \cdot M = |I|/4 \ .$$

This together with (16) implies

$$|\cup_{i\in[m]} Y_{a_i} \setminus \cup_{i=1}^{\ell} X_{a_i}| > |\cup_{i\in[m]} Y_{a_i}|/2 \ .$$

By Markov there must exist some Y_{a_t} with

$$|Y_{a_t} \setminus \cup_{i=1}^{\ell} X_{a_i}| \geq |Y_{a_t}|/2$$

as claimed. □

6 Conclusions

In this work we showed that existing time-memory trade-offs for inverting functions can be overcome, if one relaxes the requirement that the function is efficiently computable, and just asks for the function table to be computed in (quasi)linear time. We showed that such functions have interesting applications towards constructing proofs of space. The ideas we introduced can potentially also be used for related problems, like memory-bound or memory-hard functions.

Acknowledgements

Hamza Abusalah, Joël Alwen, and Krzysztof Pietrzak were supported by the European Research Council, ERC consolidator grant (682815 - TOCNeT).

Leonid Reyzin gratefully acknowledges the hospitality and support of IST Austria, where much of this work was performed. He was also supported, in part, by US NSF grants 1012910, 1012798, and 1422965.

A Proof of Lemma 1 following [DTT10]

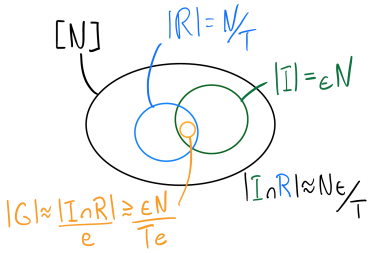

Fig. 2. The different subsets of the input domain of $f : [N] \to [N]$. R is a random subset of size N/T, I are the values x where $A^f_{\mathsf{aux}}(f(x)) = x$. G is the subset of $I \cap R$ of x where $A^f_{\mathsf{aux}}(f(x))$ makes no oracle queries to values in R (except for the last query x).

In this section we sketch the proof of Theorem 1 following the proof from [DTT10], just marginally adapting it so it applied to functions not just permutations. Let

$$I = \{x \ : \ A^f_{\mathsf{aux}}(f(x)) = x\} \quad , \quad J = f(I) = \{y \ : \ f(A^f_{\mathsf{aux}}(y)) = y\}$$

For $x \in I$ let $q(f(x))$ denote the queries made by $A^f f_{\mathsf{aux}}(f(x))$ but without the final query x. By assumption $|I| = \epsilon N$. Pick a random subset $R \subset [N]$ of size $|R| = N/T$ (for this use the randomness ρ given to the encoding). Let G denote

the set of x's in R where A^f_{aux} on input $f(x)$ finds x and makes no queries in R, i.e.,

$$G = \{x \in I \cap R \text{ and } q(f(x)) \cap R = \emptyset\}$$

The expected size of $I \cap R$ is $\epsilon N/T$ (as for any $x \in I$, $\Pr_R[x \in R] = 1/T$). For any $x \in I$, the probability that $q(f(x)) \cap R = \emptyset$ is at least $(1 - 1/T)^T \approx 1/e > 1/e$, so the expected size of G is $> \epsilon N/eT$. In this proof sketch we assume the sets have exactly their expected size, i.e.,

$$|R| = N/T \quad , \quad |I| = \epsilon N \quad , \quad |I \cap R| = \epsilon N/T \quad , \quad |G| = \epsilon N/eT$$

The encoding of $\mathsf{Enc}(\rho, f)$ now contains (the random coins r are used to sample the set R)

aux: The auxiliary input of size S.
$E_{f([N]-R)}$: The list of values $f([N] - R)$ using $(N - |R|) \log N$ bits.
$E_{\{f(G)\}}$: An encoding of the set $\{f(G)\} = \{f(x) \ : \ x \in G\}$. As $\{f(G)\}$ is ϵ/Te dense in N, so by Fact 2 this requires only $|G|(\log(e^2T/\epsilon))$ bits.
$E_{f(R-G)}$: The list of values $f(R - G)$ using $(|R| - |G|) \log N$ bits.

Overall, the encoding size is $S + |G|(\log(e^2/T\epsilon)) + (N - |G|) \log N$

$$N \log N + S - \frac{\epsilon N}{eT}(\log N - \log(e^2/T\epsilon))$$

The decoding $\mathsf{Dec}(\rho, [\mathsf{aux}, E_{f(G)}, E_{f([N]-R)}, E_{f(R-G)}])$ is straight forward

- Invoke $A^f_{\mathsf{aux}}(.)$ on all $y \in \{f(G)\}$, answering all oracle queries (except the last one) using the mapping $[N] - R \to f([N] - R)$. The last query x of $A^f_{\mathsf{aux}}(y)$ can be recognized as it's the first query not in $[N] - R$, and we learn that $f(x) = y$.
- Output $f([N])$, which can be computed as the lists G, R and $f(G), f(R - G), f([N] - R)$ are all known.

References

[BBS06] Elad Barkan, Eli Biham, and Adi Shamir. Rigorous bounds on cryptanalytic time/memory tradeoffs. pages 1–21, 2006.

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. pages 585–605, 2015.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. pages 139–147, 1993.

[DTT10] Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and PRGs. pages 649–665, 2010.

[FN91] Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. pages 534–541, 1991.

[GT00] Rosario Gennaro and Luca Trevisan. Lower bounds on the efficiency of generic cryptographic constructions. pages 305–313, 2000.

[Hel80] Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Information Theory*, 26(4):401–406, 1980.

[PPK+15] Sunoo Park, Krzysztof Pietrzak, Albert Kwon, Joël Alwen, Georg Fuchsbauer, and Peter Gaži. Spacemint: A cryptocurrency based on proofs of space. Cryptology ePrint Archive, Report 2015/528, 2015. http://eprint.iacr.org/2015/528.

[RD16] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. pages 262–285, 2016.

[Wee05] Hoeteck Wee. On obfuscating point functions. pages 523–532, 2005.

[Yao90] Andrew Chi-Chih Yao. Coherent functions and program checkers (extended abstract). pages 84–94, 1990.