Proof Systems for Sustainable Decentralized Cryptocurrencies

by

Hamza Abusalah

A thesis presented to the Graduate School of the Institute of Science and Technology Austria in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Klosterneuburg, Austria.

September, 2018.

Abstract

A proof system is a protocol between a prover and a verifier over a common input in which an honest prover convinces the verifier of the validity of true statements. Motivated by the success of decentralized cryptocurrencies, exemplified by Bitcoin, the focus of this thesis will be on proof systems which found applications in some sustainable alternatives to Bitcoin, such as the Spacemint and Chia cryptocurrencies. In particular, we focus on proofs of space and proofs of sequential work.

Proofs of space (PoSpace) were suggested as more ecological, economical, and egalitarian alternative to the energy-wasteful proof-of-work mining of Bitcoin. However, the state-of-the-art constructions of PoSpace are based on sophisticated graph pebbling lower bounds, and are therefore complex. Moreover, when these PoSpace are used in cryptocurrencies like Spacemint, miners can only start mining after ensuring that a commitment to their space is already added in a special transaction to the blockchain.

Proofs of sequential work (PoSW) are proof systems in which a prover, upon receiving a statement χ and a time parameter T, computes a proof which convinces the verifier that T time units had passed since χ was received. Whereas Spacemint assumes synchrony to retain some interesting Bitcoin dynamics, Chia requires PoSW with unique proofs, i.e., PoSW in which it is hard to come up with more than one accepting proof for any true statement.

In this thesis we construct simple and practically-efficient PoSpace and PoSW. When using our PoSpace in cryptocurrencies, miners can start mining on the fly, like in Bitcoin, and unlike current constructions of PoSW, which either achieve efficient verification of sequential work, or faster-than-recomputing verification of correctness of proofs, but not both at the same time, ours achieve the best of these two worlds.

About the Author

Hamza Abusalah completed a B.Sc. in Computer Systems Engineering at Birzeit University and an M.Sc. in Software Systems Engineering at RWTH Aachen University before joining IST Austria to pursue his doctorate studies in cryptography.

Acknowledgments

I would like to start by thanking my supervisor Krzysztof Pietrzak for his mentoring and generous support over the years, where his door has always been open and he maintained a healthy and stimulating research environment that eventually gave rise to this thesis.

The results in this thesis are jointly worked out with Joël Alwen, Bram Cohen, Chethan Kamath, Danylo Khilko, Karen Klein, Krzysztof Pietrzak, Leonid Reyzin, and Michael Walter. I thank them all for their contributions.

Besides, I would like to thank Georg Fuchsbauer for his guidance throughout our fruitful collaborations, Dennis Hofheinz for an interesting summer internship, and Eike Kiltz for introducing me to cryptography and supporting my research endeavors.

I would also want to express my gratitude to Krishnendu Chatterjee and Dennis Hofheinz for serving on my Ph.D. committee.

IST Austria, with its amazing people and diverse scientific and social events, made my research such a pleasant and memorable experience. The research in this thesis was carried out under the financial support of the European Research Council, ERC Starting (259668-PSPC) and Consolidator (682815-TOCNeT) Grants.

List of Publications

The following papers are part of my Ph.D. work, and this thesis is based on the first two:

Reversible Proofs of Sequential Work.
In submission.
Coauthored with: Chethan Kamath, Karen Klein, Krzysztof Pietrzak, Michael Walter.

Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space.
Appeared in: ASIACRYPT (2) 2017: 357-379.
Coauthored with: Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, Leonid Reyzin.

Offline Witness Encryption.
Appeared in: ACNS 2016: 285-303.
Coauthored with: Georg Fuchsbauer, Krzysztof Pietrzak.

Constrained PRFs for Unbounded Inputs with Short Keys.
Appeared in: ACNS 2016: 445-463.
Coauthored with: Georg Fuchsbauer.

Constrained PRFs for Unbounded Inputs.
Appeared in: CT-RSA 2016: 413-428.
Coauthored with: Georg Fuchsbauer, Krzysztof Pietrzak.

Contents

List of Figures

1. Introduction

The blueprint for decentralized cryptocurrencies was laid out by Bitcoin [Nak08]. The Bitcoin protocol implements a robust public transaction ledger by securely maintaining a distributed data structure called a blockchain.

A public transaction ledger, which is a public record of transactions, is robust if it guarantees that transactions get eventually added to it, and once a set of transactions is added, it is hard to change the transactions or their chronological order, a property needed to guarantee that a coin can not be spent more than once, a problem known as double spending.

A blockchain is a data structure maintained and extended in a decentralized and distributed manner by a network of system participants called miners. We assume that miners receive transactions from a peer-to-peer network in rounds. To extend the blockchain, a miner bundles some of these transactions into a new block, and produces and publishes to the network a publicly-verifiable proof attesting the validity of its block, and hence extending the blockchain. Honest miners add valid blocks to the longest chain in the blockchain, while the behavior of adversarial miners is arbitrary, and in particular, they may try to fork the blockchain.

Informally, a blockchain is secure if it is guaranteed to grow, and it grows in such a way that ignoring the blocks produced in the last few rounds, honest miners have the same view of the blockchain and their view is an actual chain, and moreover, the honest miners are assured that the ratio of adversarial blocks in such a chain is locally bounded. (For a formal treatment of these properties, see [GKL15].)

It is clear that secure blockchains give rise to robust public transaction ledgers. The challenge a cryptocurrency is then faced with is maintaining a secure blockchain. Bitcoin, for example, uses proofs of work as the underlying publicly-verifiable proofs needed to extend the blockchain, and assuming the majority of mining power is honest, the blockchain is provably secure [GKL15].

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

In the context of Bitcoin, the statement χ is determined by the blockchain and the new block of transactions, and miners, playing the role of provers $\mathcal{P}$, extend the blockchain by exhaustively searching for s such that $H(s, \chi)$ starts with a number of t zeros, where t is a dynamic parameter determined based on the rate of added blocks to the blockchain.

Proofs of space (PoSpace) [DFKP15] were suggested as more ecological, economical, and egalitarian alternative to the energy-wasteful PoW mining of Bitcoin. However, the state-of-the-art constructions of PoSpace [DFKP15, RD16] are based on sophisticated graph pebbling lower bounds, and are therefore complex. Moreover, when these PoSpace are used in cryptocurrencies like Spacemint [PPK+15], miners can only start mining after ensuring that a commitment to their space is already added in a special transaction to the blockchain.

Proofs of sequential work (PoSW) [MMV13] are proof systems in which a prover, upon receiving a statement χ and a time parameter T, computes a proof which convinces the verifier that T time units had passed since χ was received. Whereas Spacemint assumes synchrony to retain some interesting Bitcoin dynamics, Chia [chi18] requires PoSW with unique proofs, i.e., PoSW in which it is hard to come up with more than one accepting proof for any true statement.

In this thesis we construct simple and practically-efficient PoSpace and PoSW. When using our PoSpace in cryptocurrencies, miners can start mining on the fly, like in Bitcoin, and unlike current constructions of PoSW, which either achieve efficient verification of sequential work [MMV13, CP18], or faster-than-recomputing verification of correctness of proofs [LW17], but not both at the same time, ours achieve the best of these two worlds.

1.1 Proofs of Space

A proof of space (PoSpace) [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, 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 is not much more, quasilinear in N at most. On the other hand $\mathcal{V}$ 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 is 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 $\tilde{\mathcal{P}}$ which 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.

Existing constructions of PoSpace [DFKP15, RD16] are based on pebbling lower bounds for graphs. These PoSpace 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, these PoSpace have a drawback that make them more difficult to use as a replacement for PoW in blockchains. Concretely, the initialization phase requires two messages: the first message is sent from the verifier to the prover specifying a random function f, and the second message is a "commitment" from the prover to the verifier.[1]

If such a pebbling-based PoSpace 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 tricky. In the PoSpace-based cryptocurrency Spacemint [PPK+15], this is solved by having a miner put this commitment into the blockchain itself before it 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.

A simple idea for constructing a PoSpace 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.[2] Then, during the proof phase, to convince the verifier that he really stores this table, the prover must invert f on a random challenge. Unfortunately such an approach fails to give any meaningful security guarantees due to existing time-memory trade-offs.

In particular, Hellman [Hel80] showed that any *permutation* over a domain of size N

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

[2] 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 $f(x) = H(s, x)$.

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.

As a first contribution of this thesis, we rescue the simple approach of constructing PoSpace based on inverting random functions by constructing functions that bypass Hellman's attack and entertain more favorable space-time tradeoffs. The resulting PoSpace is simple and more efficient than existing solutions [DFKP15, RD16], and furthermore, when used in PoSpace-based cryptocurrencies, space miners need not wait until their commitment to their space is added to the blockchain to start mining.

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 PoSpace 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 $S^2 \cdot T \in \Omega(\epsilon^2 N^2)$.

1.2 Proofs of Sequential Work

Publicly verifiable proofs of sequential work (PoSW) introduced by Mahmoody, Moran, and Vadhan [MMV13] are proof systems in which a prover, upon receiving a statement χ and a time parameter T, computes a proof $\phi(\chi, T)$ which is efficiently and publicly verifiable. The proof can be computed in T sequential steps, but not much less, even by a malicious party having large parallelism. A PoSW thus serves as a proof that T units of time had passed since χ was received.

As possible applications for PoSW, [MMV13] lists universally verifiable CPU benchmarks and non-interactive time-stamping. More recently, there has been a renewed interest in PoSW as they form building blocks for more sustainable blockchains. Chia (`chia.net`) is an emerging cryptocurrency that uses both PoSpace and PoSW: To create a block, a space miner – using disk space it has previously initialized for mining – gen-

erates and publishes a PoSpace σ (the challenge for the proof comes from the previous block), which is then assigned a quality $q(\sigma)$. A block is a tuple of such a PoSpace and a PoSW whose challenge is σ and its time parameter is $q(\sigma)$. The rationale is that the proofs of space with best quality are likely to get extended to full blocks first, and thus will end up in the blockchain.

The construction of Mahmoody, Moran, and Vadhan [MMV13] is not practical as a prover needs not only T sequential time steps but also linear in T space to compute a proof. Cohen and Pietrzak [CP18] resolved this issue by constructing a PoSW where the prover requires just $\log(T)$ space.

A necessary property for blockchain applications of PoSW which is not achieved by the constructions of [MMV13, CP18] is "uniqueness", which means it is not possible to compute more than one accepting proof for the same statement.

A simple PoSW construction that is unique is a hash chain, where on input $x = x_0$ one outputs as proof x_T which is recursively computed as $x_i = H(x_{i-1})$ for a hash function H. This is a terrible PoSW as verification requires also T hashes. At least one can parallelize verification by additionally outputting some q intermediate values $x_0, x_{T/q}, x_{2T/q}, \ldots, x_T$. Now the proof can be verified in T/q time assuming one can evaluate q instantiations of H in parallel: for every $i \in [q]$, verify that $H^{T/q}(x_{(i-1)T/q}) = x_{iT/q}$.

Lenstra and Weselowski [LW17] suggest a construction called "sloth", which basically is a hash chain but with the additional property that it can be verified with a few hundred times less computation than what is required to compute it. The construction is based on the assumption that computing square roots in a field $\mathbb{F}_p$ of size p is around $\log(p)$ times slower than the inverse operation, which is just squaring. A typical value would be $\log(p) \approx 1000$.

As a second contribution of this thesis, we construct a new PoSW in the random permutation model which is almost as simple and efficient as [CP18]. Our construction is based on skip lists, and (unlike [CP18] but like [LW17]) has the property that generating the PoSW is a reversible computation. This property allows us to "embed" sloth in this PoSW and the resulting object is a PoSW where verifying sequential work is as efficient as in [CP18], while verifying (the stronger property) that the correct output has been computed is as efficient as in [LW17].

Now assume a malicious party wants to disrupt the mining process by flooding the network with wrong and/or malleated proofs (wrong means they won't pass verification, malleated means they pass verification, but are not computed using the honest PoSW algorithm). If one uses the PoSW from [CP18], wrong proofs are not a problem as they can be rejected extremely efficiently. But malleable proofs are a problem, as those cannot

be detected much more efficiently than basically recomputing the entire PoSW. On the other hand, when using sloth there are no malleated proofs (only the correct proof will pass verification), but rejecting wrong proofs is significantly more expensive than that of [CP18], though still much cheaper than computing the proof. Using our construction one can use the efficient verification to reject all wrong proofs as in [CP18], and only when one observes two or more distinct proofs that pass this efficient verification one falls back on the sloth-like verification procedure.

2. Proofs of Space

2.1 Overview

A proof of space (PoSpace) 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, 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 is 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.

A simple idea for constructing a PoSpace 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.[1] Then, during the proof phase, to convince the verifier that he really stores this table, the prover must invert f on a random challenge. Unfortunately such an approach fails to give any meaningful security guarantees due to existing time-memory trade-offs.

In particular, Hellman [Hel80] 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

[1] 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 $f(x) = H(s, x)$.

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.

2.1.1 Our Results

We rescue the simple approach of constructing PoSpace based on inverting random functions by constructing functions that bypass Hellman's attack and entertain more favorable space-time tradeoffs. The resulting PoSpace is simple and more efficient than existing solutions [DFKP15, RD16], and furthermore, when used in PoSpace-based cryptocurrencies, space miners need not wait until their commitment to their space is added to the blockchain to start mining.

Constructing PoSpace from inverting random functions seemed impossible [DFKP15] due to Hellman's time-memory trade-offs [Hel80]. For Hellman's attacks to apply, one needs to be able to evaluate the function efficiently in forward direction. However, we observe that for functions to be used in the simple PoSpace 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[2] – 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

[2] If f is a function, we do not 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 PoSpace in the same way as if f were a permutation.

like SHA-3 or (truncated) AES,[3] whereas we do not have good candidates for suitable permutations (at the very least f needs to be one-way; and, unfortunately, all 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

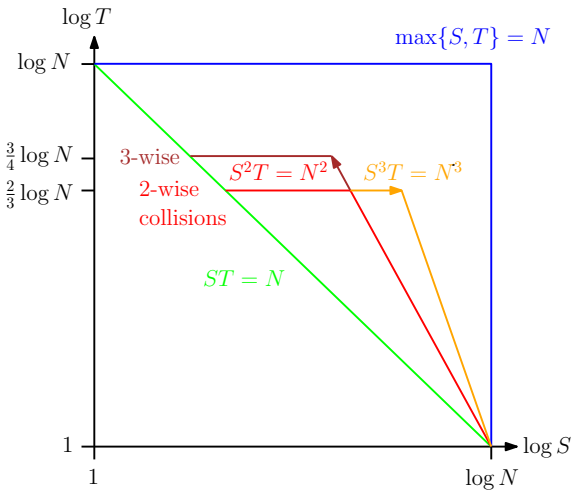

Figure 2.1: Illustration of lower bounds. Green: the $ST = \Omega(N)$ lower bound for inverting random permutations or functions. Blue: the ideal bound where either T or S is $\Omega(N)$ as achieved by the pebbling-based PoSpace [DFKP15, RD16] (more precisely, the bound approaches the blue line for large N). Red: the lower bound $S^2T = \Omega(N^2)$ for $T \leq N^{2/3}$ for our most basic construction as stated in Theorem 2. Brown: 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 brown arrow shows how the bound improves by going from $t = 2$ to $t = 3$. Orange: 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 orange arrow shows how the bound improves by going from $k = 2$ to $k = 3$.

the lower bound to $S^kT \in \Omega(\epsilon^k N^k)$ by "nesting" the construction; in the first iteration of this nesting one replaces the inner function f with g_f.[4] These lower bounds are illustrated in Figure 2.1.

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

[4] 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.

We do not give a proof for the general construction, as the proof for the general construction does not require any new ideas, but just gets more technical. We also expect the basic construction to be already sufficient for constructing a secure PoSpace. 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,[5] we expect this attack to only be of concern for extremely large N.[6]

A caveat of our lower bound is that it only applies if $T \leq N^{2/3}$. We do not 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 $x_1 < x_2 < \ldots < x_{t-1}$.

2.1.2 A Simple PoSpace that Fails

Probably the first candidate for a PoSpace 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 $y = f(x)$.[7]

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

[5] Informally, nesting Hellman's attack works as follows. Note that if we could efficiently evaluate $g_f(\cdot)$, 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 $g_f(\cdot)$. 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 $S^{2k}T \in O(N^{2k})$.

[6] 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.

[7] 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.

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 PoSpace. 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 PoSpace 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'.

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.[8]

When used as a replacement for PoW in cryptocurrencies, neither the soundness nor the completeness error are an issue. If this PoSpace is to be used in a context 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.[9]

When constructing a PoSpace from a function with a domain of size N, the space the honest prover requires is around $N \log N$ bits for the simple PoSpace 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

[8] 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 $[N]$.

[9] 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.

trade-offs we can prove on the hardness of inverting the underlying function translate directly to the security of the PoSpace.

2.1.3 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)

$$S \cdot T \in \tilde{O}(N) \quad \text{e.g. when} \quad S = T \approx N^{1/2} \ . \tag{2.1}$$

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

$$S^2 \cdot T \in \tilde{O}(N^2) \quad \text{e.g. when} \quad S = T \approx N^{2/3} \ . \tag{2.2}$$

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

$$S^3 \cdot T \in \tilde{O}(N^3) \quad \text{e.g. when} \quad S = T \approx N^{3/4} \ . \tag{2.3}$$

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 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 do not 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 $x_{(i-1)T}$.[10] If we do not 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.

[10] 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 $x_{(i-1)}T$.

2.1.4 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 does not 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 $\sigma_f(h_i(\cdot))$.

2.1.5 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 (2.1) 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 (2.2). 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.

2.1.6 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.

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 is 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 Section 2.2 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 $p(G)$ and answering all oracle queries using the mapping $[N] - G \to p([N] - G)$ where 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 is 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 proof of De et al., 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.

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 does not 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.[11] 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 is 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 Section 2.5, which implies that it is always possible to find a sequence of inputs to $\mathcal{A}^{f,g}_{\mathsf{aux}}$ such that this is the case. Concretely, we can always find a sequence of inputs

[11] Note that for the adversary as specified by Hellman's attack against permutations as outlined in Section 2.1.3 we do have $T_g = T$, which is not surprising given that for permutations the De at al. lower bound matches Hellman's attack.

such that at least $T_g \cdot \epsilon N/32T$ values can be compressed.[12]

2.2 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*

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

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* $\log \binom{n}{\epsilon n} \leq \epsilon n \log(e/\epsilon)$.

2.3 A Lower Bound for Functions

The following theorem is basically from [DTT10], but stated for functions rather than 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*

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

then

$$T \cdot S \in \Omega(\epsilon N) \ . \tag{2.4}$$

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

Theorem 1 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 (2.6) this implies (2.4). Note that the extra assumption that $T \leq \epsilon N/40$ in the lemma below does not matter, as if it is 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 $T_g = T$.

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 $|\mathsf{aux}| = S$*

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

then

$$\mathsf{Pr}_{\rho \leftarrow \{0,1\}^r} \left[\mathsf{Dec}(\rho, \mathsf{Enc}(\rho, \mathsf{aux}, f)) = f \right] \geq 0.9 \tag{2.5}$$

and the length of the encoding is at most

$$|\mathsf{Enc}(\rho, \mathsf{aux}, f)| \leq \underbrace{N \log N}_{=|f|} - \frac{\epsilon N}{2T_g} + S + \log(N) \tag{2.6}$$

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

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 16 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

$$|G| \geq \epsilon N/2T_g \ . \tag{2.7}$$

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 $|G| = |B|/T_g \geq |J|/2T_g = \epsilon N/2T_g$.

1: **Input:** $\mathcal{A}$, aux, randomness ρ and a function $f : [N] \to [N]$ to compress.
2: Initialize: $B, G := \emptyset, c := -1$
3: Throughout we identify $[N]$ with $\{0, \ldots, N-1\}$.
4: Pick a random permutation $\pi : [N] \to [N]$ (using random coins from ρ)
5: 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.
6: For $i = 0, \ldots, N-1$ define $y_i := \pi(i)$. ▷ Randomize the order
7: For $y \in J$ let the list $q(y)$ contain all queries made by $\mathcal{A}^f(y)$ except the last query (which is x s.t. $f(x) = y$).
8: **while** $|B| < |J|/2$ **do** ▷ While the bad set contains less than half of J
9: $\quad c := \min\{c' > c \ : \ y_{c'} \in \{J \setminus B\}\}$ ▷ Increase c to the next y_c in $J \setminus B$
10: $\quad G := G \cup y_c$ ▷ Add this y_c to good set
11: $\quad$ Invoke $\mathcal{A}^f_{\mathsf{aux}}(y_c)$ and observe its queries
12: $\quad B := B \cup (f(q(y_c)) \cap J)$ ▷ Add spoiled queries to bad set
13: **end while**
14: 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 $(q'_1, q'_2) = ((1,2,3),(4,5))$.
15: Let $G^{-1} = \{\mathcal{A}^f_{\mathsf{aux}}(y) \ : \ y \in G\}$
16: 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.

Algorithm 1: Enc

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 $J \setminus B$ 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 do not 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.

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 $\mathcal{A}^f_{\mathsf{aux}}(\cdot)$ 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 $f([N])$

Algorithm 2: Dec

$(|q'_1|, \ldots, |q'_{|G|}|)$: 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 (2.7) we get

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

as claimed. □

2.4 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*

$$T \leq (N/4e)^{2/3} \tag{2.8}$$

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*

$$\mathsf{Pr}_{y \leftarrow [N]} \left[g_f(A^{f,g}_{\mathsf{aux}}(y)) = y \right] \geq \epsilon \tag{2.9}$$

then

$$TS^2 \in \Omega(\epsilon^2 N^2) \ . \tag{2.10}$$

Now Theorem 2 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*

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

then

$$\mathsf{Pr}_{\rho \leftarrow \{0,1\}^r} \left[\mathsf{Dec}_g(\rho, f, \mathsf{Enc}_g(\rho, \mathsf{aux}, f, g)) = g \right] \ \geq \ 0.9 \tag{2.11}$$

$$\mathsf{Pr}_{\rho \leftarrow \{0,1\}^r} \left[\mathsf{Dec}_f(\rho, g, \mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)) = f \right] \ \geq \ 0.9 \ . \tag{2.12}$$

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

$$|\mathsf{Enc}_g(\rho, \mathsf{aux}, f, g)| \ \leq \ \underbrace{N^2 \log N}_{=|g|} - \frac{\epsilon N}{2T_g} + S + \log N \tag{2.13}$$

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

$$|\mathsf{Enc}_f(\rho, \mathsf{aux}, f, g)| \ \leq \ \underbrace{\log N!}_{=|f|} - \frac{\epsilon N T_g}{64T} + S + \log N \ . \tag{2.14}$$

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 (2.13) 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 (2.12) 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 (2.9) holds. By a union bound for at least a 0.8 fraction of the ρ, (2.11) and (2.12) 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 (2.13) 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 (2.10).

- If $T_g \geq \sqrt{T}$ then we use (2.14) and Fact 1 and again get $S \geq \epsilon N T_g/64T - \log N - \log(1/0.8)$ which implies (2.10) as $T_g \geq \sqrt{T}$.

□

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 $E_{g_f} \leftarrow \mathsf{Enc}(\mathcal{A}, g_f, \mathsf{aux}, \rho)$
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 3: Enc_g

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

Algorithm 4: Dec_g

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 $f([N] - \{L_f^{-1} \cup C_f^{-1}\})$ which were neither in the list L_f nor compressed.

Algorithm 5: Enc_f

Input: $\mathcal{A}, \rho, \mathsf{aux}, g$ and encoding $(G_f, L_f, T_f, f([N] - \{L_f^{-1} \cup C_f^{-1}\}))$ 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 $f([N])$

Algorithm 6: Dec_f

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. The adversary $\mathcal{A}^{f,g}_{\mathsf{aux}}(\cdot)$ can make up to T queries in total to its oracles $f(\cdot)$ and $g(\cdot)$. 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 do not 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 (2.13) and (2.14).

Now (2.13) 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 (2.14). Recall that Enc (as used inside Enc_g) defines a set G such that for every $y \in G$ we have that $A^{f,g}_{\mathsf{aux}}(y)$ inverts, i.e., $g_f(A^{f,g}_{\mathsf{aux}}(y)) = y$ and it never makes a g_f query x where $g_f(x) \in G$. Recall that T_g in (2.13) 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

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

And if $\log N \leq T_g/8$, we get (2.14), i.e.,

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

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 Section 2.5. 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.[13] 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 $|G_f| \geq \epsilon N/8T$.[14]

[13] 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 $|X_i| \leq T$.

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

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 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[15], 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 this concludes the proof.

□

2.5 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 $Y_1, \ldots, Y_n \subset [N]$*

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

[15] 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.

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

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

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

$$Z_{b_j} = Y_{b_j} \setminus \cup_{k<j} X_{b_k} \tag{2.15}$$

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.

$$\left|\cup_{i\in[m]} Y_{a_i}\right| > |I|/2 \ . \tag{2.16}$$

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 (2.16) 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. □

2.6 Open Problems

We constructed a family of functions from $[N]$ to $[N]$ such that for any adversary that inverts a uniformly sampled function from this family on an ϵ fraction of images, using at most $T \leq (N/4e)^{2/3}$ forward-direction oracle queries and an advice of size S bits, it must be the case that

$$TS^2 \in \Omega(\epsilon^2 N^2) \ . \tag{2.17}$$

It would be very interesting to prove (2.17) for the full range of parameters, namely for all $T \leq \alpha N$ for some constant $\alpha \in (0, 1)$. The restriction on T in our result seems inherent to the encoding we use and we see no reason why it can't be improved.

Furthermore, we argued that by nesting our construction a constant number of times k, the bound in (2.17) becomes

$$TS^k \in \Omega(\epsilon^k N^k) \ . \tag{2.18}$$

However, the best attacks that we are aware of are still Hellman's attacks, and for a nesting parameter k, such attacks result in an upper bound that satisfies

$$TS^{2k} \in O(\epsilon^{2k} N^{2k}) \ . \tag{2.19}$$

The natural open question here is figuring out the optimal trade-off, by improving either bounds.

3. Proofs of Sequential Work

3.1 Overview

Timed-release cryptography was envisioned by May [May93] and realized by Rivest, Shamir, and Wagner [RSW00] in the form of a "time-lock puzzle". For some time parameter T, such a puzzle can be efficiently sampled together with a solution. However, solving it requires T sequential computations, and this holds even for parties aided with massive parallelism. In other words, there are no "shortcuts" to the solution. The application envisioned in [RSW00] was "sending a message to the future": generate a puzzle, derive a symmetric key from the solution, encrypt your message using that key, and release the ciphertext and the puzzle.

The construction put forward in [RSW00] is in the RSA setting: the puzzle is a tuple (N, x, T), where $N = p \cdot q$ is an RSA modulus and $x \in Z_N^*$ a group element, and the solution to the puzzle is $x^{2^T} \bmod N$. Although the solution can be computed efficiently *if* the factorization of N is known, it was conjectured to require T sequential squarings given only N. The assumption that underlies the soundness of the [RSW00] time-lock puzzle is rather non-standard (which is basically that the puzzle is sound, i.e., there's no shortcut in computing the solution) and it is an open problem to come up with constructions under more standard assumptions.

In a negative result, Mahmoody, Moran, and Vadhan [MMV11] show that there's no black-box construction of a time-lock puzzle in the random oracle model. In a subsequent work the same authors [MMV13] propose and construct publicly verifiable proofs of sequential work.

Publicly verifiable proofs of sequential work (PoSW) are proof systems in which a prover, upon receiving a statement χ and a time parameter T, computes a proof $\phi(\chi, T)$ which is publicly verifiable. The proof can be computed in T sequential steps, but not much less, even by a malicious party having large parallelism: soundness implies some failure probability of malicious provers dedicating $\alpha \cdot T$ parallel time. Verification is considered efficient if it can be done in time poly-logarithmic in T and a security parameter.

(We refer the reader to [MMV13] for more formal definition. Due to lack of consensus of what the right definitions are, in this work we choose to leave the definitions informal, however we prove precise and unambiguous statements.)

Even though PoSW seem related to time-lock puzzles, they are not directly comparable. In particular, in a PoSW it is not possible to sample the solution together with an instance. On the positive side, PoSW can be constructed in the random oracle model (or under a standard model assumption on hash functions called “sequentiality"). They are publicly verifiable and sampling the challenge is public-coin.

The first construction of efficiently and publicly verifiable PoSW is due to Mahmoody, Moran, and Vadhan [MMV13]. The construction is however not of practical value as a prover needs not only T sequential time steps but also linear in T space to compute a proof. Cohen and Pietrzak [CP18] resolved this issue by constructing PoSW where proofs can be computed in $\log(T)$ space and T time steps.

A necessary property for blockchain applications of PoSW which is not achieved by the constructions of [MMV13, CP18] is “uniqueness", which means it is not possible to compute more than one accepting proof for the same statement.

A simple PoSW construction that is unique is a hash chain, where on input $x = x_0$ one outputs as proof x_T which is recursively computed as $x_i = H(x_{i-1})$ for a hash function H. This is a terrible PoSW as verification requires also T hashes. At least one can parallelize verification by additionally outputting some q intermediate values $x_0, x_{T/q}, x_{2T/q}, \ldots, x_T$. Now the proof can be verified in T/q time assuming one can evaluate q instantiations of H in parallel: for every $i \in [q]$, verify that $H^{T/q}(x_{(i-1)T/q}) = x_{iT/q}$.

Lenstra and Weselowski [LW17] suggest a construction which is basically a hash chain but with the additional property that proofs can be verified with a few hundred times less computation than what is required to compute it. Their construction, dubbed“sloth” is based on the assumption that computing square roots in a field $\mathbb{F}_p$ of size p is around $\log(p)$ times slower than the inverse operation, which is just squaring. A typical value would be $\log(p) \approx 1000$.

Their idea is to simply use a hash chain where the hash function is some permutation $\pi : \mathbb{F}_p \to \mathbb{F}_p$, where $\mathbb{F}_p$ is a finite field of size p, followed by taking a square root: that is $x_i = \sqrt{\pi(x_{i-1})}$. Verification goes as for a standard hash chain, but one computes backwards, checking $x_{i-1} = \pi^{-1}(x_i^2)$, which – assuming computing π, π^{-1} is cheap compared to squaring, and squaring is $\log(p)$ times faster than taking square roots – gives the claimed speedup of $\approx \log(p)$ compared to a simple hash chain. We will call a function like sloth – where from a given state x_i, one can compute the previous state x_{i-1} – “reversible".

In this work we construct a new PoSW in the random permutation model which is

almost as simple and efficient as [CP18]. Our construction is based on skip lists, and (unlike [CP18] but like [LW17]) has the property that generating the PoSW is a reversible computation. This property allows us to "embed" sloth in this PoSW and the resulting object is a PoSW where verifying sequential work is as efficient as in [CP18], while verifying (the stronger property) that the correct output has been computed is as efficient as in [LW17].

3.2 Constructing PoSW

3.2.1 Notation

Throughout we denote the time parameter of our construction by $N = 2^n$ with $n \in \mathbb{N}$ and assume it is a power of 2. We reserve $w, t \in \mathbb{N}$ to denote two statistical security parameters where w is the block size (say $w = 256$) and t denotes the number of challenges: a cheating prover who only makes $N(1-\epsilon)$ sequential steps (instead of N) will pass verification with probability $(1-\epsilon)^t$. For integers m, m' we denote with $[m, m'] = \{m, m+1, \ldots, m'\}$, $[m], [m]_0$ are short for $[1, m]$ and $[0, m]$.

We define $\tilde{0} = n + 1$ and for $i \geq 1, \tilde{i}$ equals 1 plus the number of trailing zeros in the binary representation of i. For example,

$$\tilde{0}, \tilde{1}, \tilde{2}, \tilde{3}, \tilde{4}, \tilde{5}, \tilde{6}, \tilde{7}, \tilde{8}, \tilde{9}, \ldots \;=\; n+1, 1, 2, 1, 3, 1, 2, 1, 4, 1, \ldots \;\; .$$

For $\sigma \in \{0,1\}^{w \cdot i}$ let $\sigma^{(j)}$ denote the jth w-bit block of σ, so $\sigma = \sigma^{(1)} \| \ldots \| \sigma^{(i)}$, and $\sigma^{(i \ldots j)}$ denote $\sigma^{(i)} \| \ldots \| \sigma^{(j)}$.

For a permutation π over ℓ bits string, we denote with $\dot{\pi}$ the function over bit stings of length $\geq \ell$ which simply applies π to the ℓ bit prefix of the input, and leaves the rest untouched. The inverse $\dot{\pi}^{-1}$ is defined similarly.

3.2.2 The Sequence σ_Π

At the core of our construction is a mapping based on the skip list data structure (see Figure 3.1). It is built from a set of permutations $\Pi = \{\pi_i\}_{i \in [N]_0}$, where each π_i is over $\{0,1\}^{w \cdot \tilde{i}}$, and defines a sequence of states $\sigma_\Pi = \sigma_0, \ldots, \sigma_N$, $\sigma_i \in \{0,1\}^{(n+1) \cdot w}$, recursively as

$$\sigma_0 = \dot{\pi}_0(0^{w \cdot (n+1)}) \quad \text{and for } i > 0 \;:\; \sigma_i = \dot{\pi}_i(\sigma_{i-1}) \quad \left(= \pi_i(\sigma_{i-1}^{(1 \ldots \tilde{i})}) \| \sigma_{i-1}^{(\tilde{i}+1 \ldots (n+1))} \right).$$

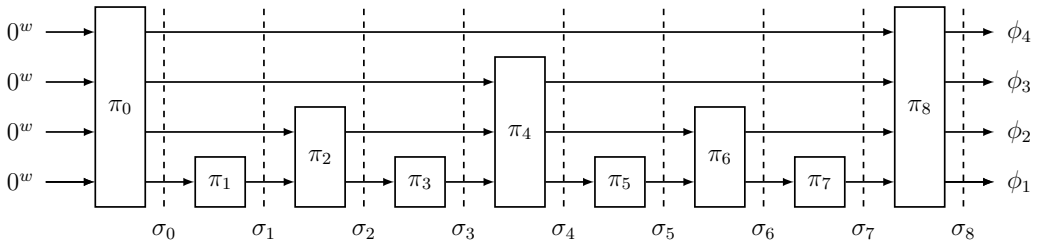

Figure 3.1: Illustration of the computation of $\sigma_\Pi = (\sigma_0, \ldots, \sigma_N)$ with $n = 3, N = 2^n = 8$. The blocks represent the permutations, whereas the dashed vertical lines represent the states. Note that the structure of the graph is the same as a skip list with four layers, where a pointer in layer $i \in \{0, 1, 2, 3\}$ points to the 2^i-th element to its right in the list.

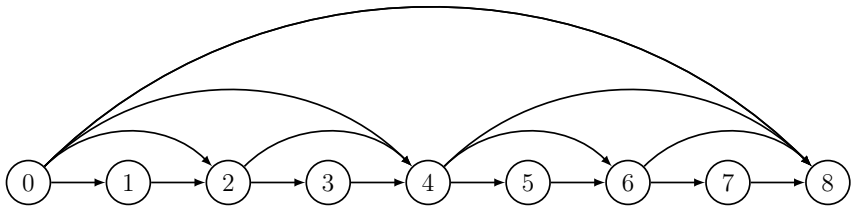

Figure 3.2: The graph G_8 that corresponds to the computation of σ_Π with $n = 3$.

3.2.3 The DAG G_N

It will be convenient to consider the directed acyclic graph (DAG)

$$G_N = (V, E)\ ,\ V = [N]_0\ ,\ E = \{(i, j) \in V^2\ : \exists k \geq 0\ : j - i = 2^k, 2^k | i\}$$

that is derived from the computation of σ_Π as follows: identify the permutation π_i with the node i and add a directed edge (i, j) if in the computation of σ_Π part of the output of π_i is piped through directly to π_j (see Figure 3.2).

For $i \in [N-1]$ we denote with $\mathsf{path}(i) \subseteq V$ the shortest path in G_N which starts at 0, ends at N, and passes through node i. For example, in Figure 3.2 $\mathsf{path}(5) = (0, 4, 5, 6, 8)$ and $\mathsf{path}(4) = (0, 4, 8)$. It is not hard to check that $|\mathsf{path}(i)| = n + 3 - \tilde{i}$, and in particular it is never longer than $n + 2$.

3.2.4 Consistent States/Paths

By construction, $\sigma_i \in \sigma_\Pi$ satisfies $\sigma_i = \dot{\pi}_{i+1}^{-1}(\sigma_{i+1})$, and more generally, for every edge $(i, j) \in E$ and $d = \min(\tilde{i}, \tilde{j})$

$$\sigma_i^{(d \ldots n+1)} = (\dot{\pi}_j^{-1}(\sigma_j))^{(d \ldots n+1)}.$$

We say two strings are consistent for (i, j) if they satisfy this condition.

Definition 1 (Consistent States/Path)**.** $\alpha_i, \alpha_j \in \{0,1\}^{(n+1)\cdot w}$ are consistent for edge $(i, j) \in E$ if with $d = \min(\tilde{i}, \tilde{j})$

$$\alpha_i^{(d\ldots n+1)} = (\dot{\pi}_j^{-1}(\alpha_j))^{(d\ldots n+1)} \ .$$

We say $\alpha'_i \in \{0,1\}^{\tilde{i}\cdot w}, \alpha'_j \in \{0,1\}^{\tilde{j}\cdot w}$ are consistent if they can be "padded" to consistent α_i, α_j as above, which is the case if

$$\alpha'^{(d)}_i = \pi^{-1}(\alpha'_j)^{(d)}.$$

We say $\{\alpha_i\}_{i\in\mathsf{path}(k)}$ are consistent with $\mathsf{path}(k)$ if for every edge $(i, j) \in \mathsf{path}(k)$ the α_i, α_j are consistent.

Note that if α_j is computed from α_i by applying $\dot{\pi}_{i+1}, \ldots, \dot{\pi}_j$ to α_i, then those α_i, α_j will be consistent with (i, j), but the converse is not true (except if $j = i + 1$).

3.2.5 PoSW Construction

The protocol between $\mathcal{P}, \mathcal{V}$ on common input $N = 2^n, w, t$ is defined as follows

1. $\mathcal{V}$ samples $\chi \leftarrow \{0,1\}^{w\cdot n}$ and sends it to $\mathcal{P}$. This χ defines a fresh set of random permutations Π (cf. Remark 1 below).
2. $\mathcal{P}$ computes $\sigma_0, \ldots, \sigma_N$ and sends $\phi = \sigma_N$ to $\mathcal{V}$.
3. $\mathcal{V}$ samples t challenges $\gamma = (\gamma_1, \ldots, \gamma_t) \leftarrow [N-1]^t$ and sends them to $\mathcal{P}$.
4. $\mathcal{P}$ sends $\{\sigma_i\}_{i\in\mathsf{path}(\gamma_j),\gamma_j\in\gamma}$ to $\mathcal{V}$ (cf. Remark 2 below).
5. $\mathcal{V}$ verifies that $\{\sigma_i\}_{i\in\mathsf{path}(\gamma_j),\gamma_j\in\gamma}$ are consistent as in Definition 1. If any check fails output reject, and output accept otherwise.

Remark 1 (Seeding Random Oracles/Permutations)**.** Ideal permutations can be constructed from random oracles [CPS08, HKT11, DSKT16] (formally, the ideal permutation model is indifferentiable from the random oracle model), so we can realize Π in the standard random oracle model.[1] Consider a fixed random oracle $\mathcal{H}(\cdot)$ about which a potential adversary has some auxiliary input (i.e., it has queried it on many inputs before,

[1] In practice, one could e.g. use χ to sample $N + 1$ AES keys $k_0, \ldots, k_N$, and then use $AES(k_i, \cdot) : \{0,1\}^{256} \to \{0,1\}^{256}$ – i.e., AES with a fixed public key – to construct π_i, where for $\tilde{i} > 1$ one would use domain extension for random permutations to extend the domain to $256 \cdot \tilde{i}$ bits.

and stored some information aux). If one samples a random seed χ and uses it as a prefix to define the function $\mathcal{H}_\chi(x) = \mathcal{H}(\chi \| x)$, this $\mathcal{H}_\chi$ – from the adversary's perspective – is a fresh random oracle as long as this seed is just a bit longer than $\log(|\mathsf{aux}|)$ [DGK17]. Thus, we can also sample a fresh Π by just sending a seed χ.

Remark 2 ($\mathcal{P}$'s Space Requirement)**.** To avoid any extra computation in step 4., $\mathcal{P}$ would need to store the entire $\sigma_\Pi = \{\sigma_i\}_{i \in [N]_0}$. By using a bit of extra computation, one can reduce the space requirement (we remark that a similar issue comes up in [CP18]). Concretely, for some $K = 2^k$, we let $\mathcal{P}$ only store σ_i where $2^k | i$, thus storing only N/K states. From this, every state σ_i can be computed making at most $K/2$ invocations to Π ($K/2$ not K as we can also compute backwards).

3.3 Security Proof

Theorem 3. *Consider a malicious prover $\tilde{\mathcal{P}}$ which*

1. *makes at most $N - \Delta$* sequential *queries to permutations in Π before sending $\phi = \sigma_N$ (in step 3 of the protocol), and*

2. *queries the permutations in Π on at most q inputs in total during execution of the protocol.*

Then $\tilde{\mathcal{P}}$ will win (i.e., make $\mathcal{V}$ output accept) with probability at most

$$\mathsf{Pr}\left[\tilde{\mathcal{P}} \text{ wins }\right] \leq \frac{2q^2(n+2)^2}{2^w} + \left(\frac{N-\Delta}{N}\right)^t . \tag{3.1}$$

Proof. We define an event bad in Definition 2 below, and can now split the probability in (3.1) to two terms

$$\mathsf{Pr}\left[\tilde{\mathcal{P}} \text{ wins }\right] \leq \mathsf{Pr}\left[\tilde{\mathcal{P}} \text{ wins and } \neg\mathsf{bad}\right] + \mathsf{Pr}\left[\mathsf{bad}\right] .$$

The theorem then follows from Lemmas 4 and 6 which independently bound these probabilities. □

Informally, the event bad holds if $\tilde{\mathcal{P}}$ makes a (forward or backward) query to a $\pi_i \in \Pi$ where some w-bit block of the output collides with some w-bit block used as input or output to some $\pi_j \in \Pi$ query made in the current or previous parallel queries. This is somewhat analogous to the "sequentiality" property which is defined for functions (not permutations) and used in [MMV13, CP18].

Definition 2 (The Event bad)**.** We describe a query to $\Pi = \{\pi_i\}_{i\in[N]_0}$ by a tuple (i, x, y, b) where $i \in [N]_0, x, y \in \{0,1\}^{\tilde{i}\cdot w}, b \in \{+,-\}$. Here $(i, x, y, +)$ denotes a forward query to π_i on input x and output $y = \pi_i(x)$, $(i, x, y, -)$ denotes an inverse query $x = \pi_i^{-1}(y)$.

Let $Q_1, Q_2, \ldots, Q_{N-\Delta}$ denote $\tilde{\mathcal{P}}$'s parallel queries. Without loss of generality we assume that $\tilde{\mathcal{P}}$ never makes a redundant query, i.e., it never makes the same query twice, and never queries $\pi_i(x)$ or $\pi_i^{-1}(y)$ if a query $(i, x, y, \star)$ has already been observed. We use $\sim$ to denote that two strings (composed of w-bit blocks) contain an identical block

$$\alpha \sim \alpha' \iff \exists i, j : \alpha^{(i)} = \alpha'^{(j)} \ .$$

The event bad holds if we have two queries $(i, x, y, b) \in Q_k, (i', x', y', b') \in Q_\ell$ where $k \le \ell$ and

$$b' = + \text{ (so } y' \text{ is the output) and } y' \sim x \text{ or } y' \sim y, \text{ or}$$
$$b' = - \text{ and } x' \sim x \text{ or } x' \sim y.$$

Lemma 4.

$$\mathsf{Pr}\left[\mathsf{bad}\right] \le \frac{2q^2(n+2)^2}{2^w} \ .$$

To prove Lemma 4 it would be convenient to prove a PRP/PRF-like switching lemma where the adversary gets oracle access to a permutation and its inverse.

Lemma 5. *Let $\pi : \{0,1\}^w \to \{0,1\}^w$ be a random permutation and consider an algorithm $\mathcal{A}^{\pi,\pi^{-1}}$ with oracle access to π and π^{-1} that makes exactly q queries in total. Assume that $\mathcal{A}$ does not repeat any queries to π nor any queries to π^{-1}, and that if it queries π at x, it does not query π^{-1} at $\pi(x)$ and vice versa. Let B_w be an oracle that on input $x \in \{0,1\}^w$ ignores x and simply returns a uniform element in $\{0,1\}^w$. Then for any event E over the output of $\mathcal{A}$, we have*

$$\mathsf{Pr}\left[\mathcal{A}^{\pi,\pi^{-1}} \in E\right] \le \mathsf{Pr}\left[\mathcal{A}^{B_w,B_w} \in E\right] + \frac{q(q-1)}{2^w}$$

where the first probability is over the choice of π and the randomness of $\mathcal{A}$, and the second over the randomness of B_w.

Lemma 5 shows that in the analysis one can replace a random permutation and its inverse oracle with random coin flips. This is very similar to the PRP/PRF switching lemma: without the inverse oracle, this would be exactly the PRP/PRF switching lemma if no queries are repeated, because then B_w is identical to a random function. One might wonder if one could simply replace both π and π^{-1} with the same random function F, since this is very similar to random coin flips. However, that does not work, because the algorithm might query π and π^{-1} at the same point x. In this case the algorithm expects

two different responses with overwhelming probability, while F would return the same value. This is why we replaced the random function by random coin flips. Equivalently, one could replace π and π^{-1} with two different, independent random functions.

of Lemma 5. Let $X = (X_1, \ldots, X_q)$ be the random variable corresponding to the responses to the queries of $\mathcal{A}^{\pi,\pi^{-1}}$ and $Y = (Y_1, \ldots, Y_q)$ the one corresponding to the responses to the queries of $\mathcal{A}^{B_w,B_w}$. We will show that the statistical distance $\Delta_{SD}(X, Y)$ is upper bounded by $\frac{q(q-1)}{2^w}$. The lemma then follows from standard properties of Δ_{SD}.

In the following, we abbreviate the conditional distributions $(X_i|X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ as $(X_i|(x_1, \ldots, x_{i-1}))$ and similarly for Y. From sub-additivity for joint distributions (a property of Δ_{SD}), we have

$$\Delta_{SD}(X, Y) \leq \sum_{i=1}^{q} \max_{x=(x_1,\ldots,x_{i-1})} \Delta_{SD}(X_i|x, Y_i|x) \ .$$

For each particular i we have

$$\Delta_{SD}(X_i|x, Y_i|x) = \frac{1}{2} \sum_{y\in\{0,1\}^w} |\mathsf{Pr}\left[X_i = y|x\right] - \mathsf{Pr}\left[Y_i = y|x\right]| \ .$$

From the definition of B_w, it is clear that $\mathsf{Pr}\left[Y_i = y|x\right] = 2^{-w}$ for all $y \in \{0,1\}^w$ and $x \in (\{0,1\}^w)^{i-1}$. For the other case, notice that any query to π or π^{-1} fixes a particular input/output pair. Accordingly, X_i is uniform among the remaining $2^w - (i-1)$ values, no matter if π or π^{-1} was queried (recall that no input/output pair is repeated). It follows that

$$\begin{aligned}\Delta_{SD}(X_i|x, Y_i|x) &= \frac{1}{2}\left[\frac{i-1}{2^w} + (2^w - (i-1))\left(\frac{1}{2^w - (i-1)} - \frac{1}{2^w}\right)\right] \\ &= \frac{i-1}{2^w}\end{aligned}$$

for any x (in particular, the maximum). Summing over all i yields the final bound. □

Note that by a simple hybrid argument, Lemma 5 also holds for families of permutations, where q is the sum over all queries and w is the minimal input/output length over all permutations. After applying Lemma 5, the proof of Lemma 4 is almost trivial.

of Lemma 4. Let $\tilde{\mathcal{P}}^B$ denote the prover where for every query $(i, x, y, +)$ we replace y with a uniformly random string $y \leftarrow \{0,1\}^{\tilde{i}\cdot w}$ and for every query $(i, x, y, -)$ we replace x with a uniformly random $x \leftarrow \{0,1\}^{\tilde{i}\cdot w}$. We apply Lemma 5 to $\tilde{\mathcal{P}}$ to obtain

$$\mathsf{Pr}\left[\mathsf{bad}\right] \leq \mathsf{Pr}\left[\mathsf{bad} \text{ occurs under } \tilde{\mathcal{P}}^B\right] + \frac{q(q-1)}{2^w} \ .$$

Under $\tilde{P}^B$, the probability that a query collides (in the sense of $\sim$, c.f. Definition 2) with a specific string in a specific previous query is at most $(n+1)^2/2^w$, since there are at

most $n+1$ blocks in each string. By union bound, the probability that a query collides with any of the previous queries is thus at most $2q(n+1)^2/2^w$, since there are two strings in each query (input and output). Applying a final union bound to all queries we get

$$\mathsf{Pr}\left[\mathsf{bad} \text{ occurs under } \tilde{\mathcal{P}}^B\right] \leq \frac{2q^2(n+1)^2}{2^w} \ ,$$

which concludes the proof. □

Lemma 6. $\mathsf{Pr}\left[\tilde{\mathcal{P}} \textit{ wins and } \neg\mathsf{bad}\right] \leq \left(\frac{N-\Delta}{N}\right)^t$

Proof. Let

$$Q = \{(i, x, y) \ : \ \tilde{\mathcal{P}} \text{ made a query } \pi_i(x) \text{ or } \pi_i^{-1}(y)\}$$

denote all the queries made by $\tilde{\mathcal{P}}$. We define a graph $G_Q = (V_Q, E_Q)$ where $V_Q = Q$, and we have an edge $((i, x, y), (i', x', y')) \in E_Q$ if the states y, x' are consistent for edge $(i, i') \in G_N$ as in Definition 1. Formally $G_Q = (V_Q, E_Q)$ where $V_Q = Q$ and

$$E_Q = \{(v, u) \in V_Q^2 : v = (i, x, y), u = (i', x', y'), i' = i + 2^\delta, 2^\delta | i, y^{(\delta+1)} = x'^{(\delta+1)}\} \ .$$

Note that the first and last query an honest prover makes while computing σ_Π are

$$\beta_0 := (0, 0^{w\cdot(n+1)}, \pi_0(0^{w\cdot(n+1)})) \quad , \quad \beta_\phi := (N, \pi_N^{-1}(\phi), \phi)$$

where $\phi = \sigma_N$. We consider a subgraph H_N of G_Q which is derived as follows. H_N contains the nodes β_0, β_ϕ (let us stress that β_ϕ is only defined by the ϕ sent by $\tilde{\mathcal{P}}$, and thus can be different from the σ_N an honest prover would send). We then recursively add vertices from G_Q to H_N, but only vertices which can potentially be used in an accepting opening of a path (step 4 and 5 in the protocol). This means we can add a node if both its "farthest" left and right neighbours are already in H_N, i.e., add $v = (i, x, y)$ to H_N if some nodes $l = (i - 2^{\tilde{i}-1}, x', y'), r = (i + 2^{\tilde{i}-1}, x'', y'')$ are already in H_N and $(l, v), (v, r) \in E_Q$. The exact procedure to compute this $H_N = \mathsf{prune}(G_Q, \beta_0, \beta_\phi)$ as well as a toy example of G_Q, H_N are given in Algorithm 7 and Figure 3.3 respectively.

We claim the following points are true *under the assumption that the event* bad *does* **not** *hold*

1. There is a path of length $|H_N|$ that starts at the source β_0 and ends at the sink β_ϕ.
2. $\tilde{\mathcal{P}}$ made at least $|H_N|$ sequential queries before sending ϕ.
3. $\tilde{\mathcal{P}}$ can output a valid $\{\sigma'_i\}_{i\in\mathsf{path}(j)}$ only for j where a node $(j, \star, \star)$ is in H_N.

Assuming points 2. and 3., the lemma can be established as follows: As $\tilde{\mathcal{P}}$ made $\leq N - \Delta$ sequential queries by assumption, by point 2. we get $|H_N| \leq N - \Delta$. Furthermore, by point 3. $\tilde{\mathcal{P}}$ will succeed to generate $\{\sigma'_i\}_{i \in \mathsf{path}(j)}$ on a random challenge j with probabiliy $\leq |H_N|/N \leq (N - \Delta)/N$, and thus with probability $\leq ((N - \Delta)/N)^t$ on t random challenges, as claimed in the statement of the lemma.

It remains to prove the three points claimed above.

1. By construction of the pruning algorithm (Alg. 7) during the update step $\mathsf{update}(\ell, r)$, unless the event bad happens, there will be at most one $v \in V$ such that $(\ell, v), (v, r) \in E$. Assume that there is another such v' then a diamond structure is formed: there are paths P_0, P_1 respectively passing through ℓ, v, r and ℓ, v', r. Such a structure triggers the event bad.

2. By the previous point, H_N is a path (with extra edges), and any two consecutive nodes in that path correspond to queries that are consistent (by the way H_N is constructed). The only way to have a sequence of $|H_N|$ consistent queries without triggering bad is to make the corresponding queries sequentially.

3. This follows by observing the following fact. If $\{\sigma'_i\}_{i \in \mathsf{path}(j)}$ is a consistent path, then all the queries corresponding to this path would have been added to H_N (as $\mathcal{V}$ checks that this path is consistent, and a consistent path would be included in H_N by definition of Algorithm 7). This argument only holds if all the queries in this path were made before $\tilde{\mathcal{P}}$ send ϕ, but $\tilde{\mathcal{P}}$ might have made some queries leading to this accepting path only after sending ϕ, in which case the path is not contained in H_N. To cover this case, note that this path must start at β_0 and end at β_ϕ (like H_N), but as it is not contained in H_N, it creates a diamond structure (as in point 1. above), which is not possible without triggering the bad event.

□

3.4 Embedding Sloth

As discussed in the introduction, we propose a reversible PoSW that is almost as efficient as the construction from [CP18] but achieves a larger time gap between the computation of the proof and the verification of correctness. To this aim, we embed the sloth hash function from [LW17] into construction 3.2.5.

The idea underlying sloth is to use the fact that the best known algorithms for computing modular square roots in a field $\mathbb{F}_p$ takes $\approx \log(p)$ sequential squarings, whereas

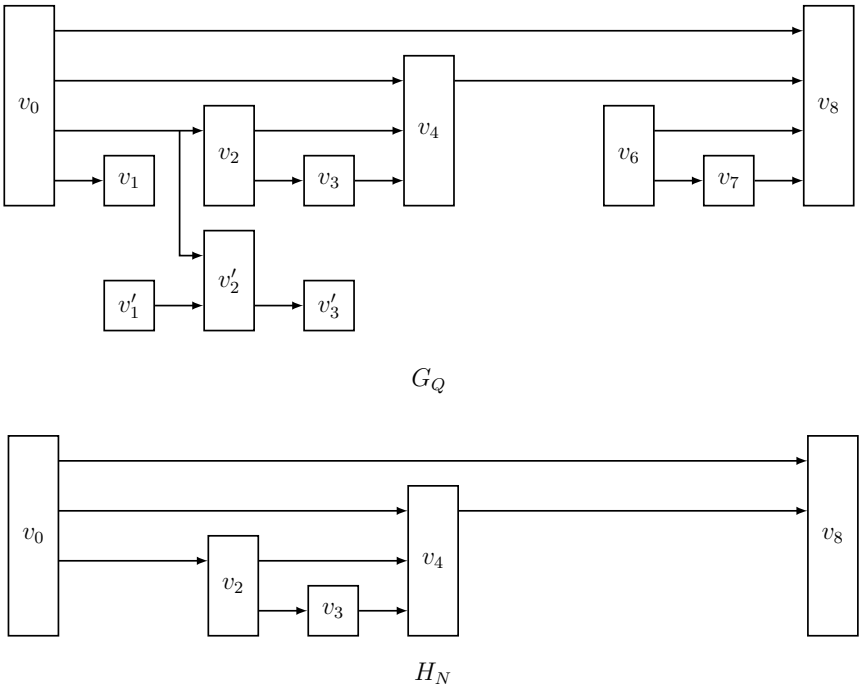

Figure 3.3: Toy example graphs for G_Q and $H_N = \mathsf{prune}(G_Q, \beta_0, \beta_\phi)$. A node v_i in H_N is a query of the form (i, x_i, y_j). In this example $\beta_0 = v_0, \beta_\phi = v_8 = (8, x_8, y_8 = \phi)$.

$\mathsf{prune}((V, E), \beta_0, \beta_\phi)$

1: $S, T = \emptyset$ ▷ Global variables; S, T are sets of nodes and edges respectively
2: **if** $(\beta_0, \beta_\phi) \in E$ **then**
3: $\quad S = S \cup \{\beta_0, \beta_\phi\}$
4: $\quad T = T \cup \{(\beta_0, \beta_\phi)\}$
5: $\quad \mathsf{update}(\beta_0, \beta_\phi)$
6: **end if**
7: **return** $(V \cap S, E \cap T)$ ▷ Return the pruned graph

$\mathsf{update}(\ell, r)$

1: **for** $v \in V : (\ell, v), (v, r) \in E$ **do** ▷ Unless bad happens there exists at most one v
2: $\quad S = S \cup \{v\}$
3: $\quad T = T \cup \{(\ell, v), (v, r)\}$
4: $\quad \mathsf{update}(\ell, v)$
5: $\quad \mathsf{update}(v, r)$
6: **end for**
7: **return**

Algorithm 7: Pruning Query Graphs

verification of the result only takes a single modular squaring. Thus, this gives a good candidate to build sloth.

For a prime $p \equiv 3 \bmod 4$, let $\mathbb{F}_p$ denote the finite field with p elements, and $\mathbb{F}_p^\times$ its multiplicative group. We represent elements of $\mathbb{F}_p$ canonically in $[0, p-1]$. If $x \in \mathbb{F}_p^\times$ is a quadratic residue, then there are two square roots $y, y' \in \mathbb{F}_p^\times$, where $y' = p - y$, one of them being even, the other odd. Let $\sqrt[+]{x}$, $\sqrt[-]{x}$ denote the (unique) even and odd square root of x, respectively. If $x \in \mathbb{F}_p^\times$ is not a quadratic residue, then $-x$ is a quadratic residue, so it makes sense to define a permutation $\rho : \mathbb{F}_p^\times \to \mathbb{F}_p^\times$ as

$$\rho(x) = \begin{cases} \sqrt[+]{x} & \text{if } x \text{ is a quadratic residue} \\ \sqrt[-]{-x} & \text{otherwise} \quad . \end{cases}$$

Its inverse is defined by

$$\rho^{-1}(x) = \begin{cases} x^2 & \text{if } x \text{ is even} \\ -x^2 & \text{otherwise} \quad . \end{cases}$$

Unfortunately, one cannot directly build a hash chain by iterating ρ since reducing modulo $p-1$ in the exponent would yield a much faster computation than sequentially computing ρ. Lenstra and Wesolowski [LW17] solve this problem by prepending an easily computable (in both directions) permutation ι on $\mathbb{F}_p^\times$ to each iteration of the square rooting function ρ. Setting $\tau = \rho \circ \iota$, the sloth function is hence defined as τ^N for some appropriate chain length N. Verification can be done backwards by the computation $(\tau^N)^{-1} = (\iota^{-1} \circ \rho^{-1})^N$, which is by a factor $\log p$ faster.

Security of sloth is proven [LW17] in the random oracle model: Assuming ι is a random permutation and that computing the square root of a random square in $\mathbb{F}_p^\times$ requires $\Omega(\log(p))$ sequential multiplications, then sloth is inherently sequential.

We now combine the ideas from [LW17] with our construction to achieve an efficient PoSW while preserving the fast verification of correctness obtained by the sloth construction. Let $\Pi = \{\pi_i\}_{i \in [N]_0}$ be a set of permutations where, for each $i \in [N]_0$, $\pi_i : \mathbb{F}_p^\times \times \{0,1\}^{w \cdot (\tilde{i}-1)}$. We define the sequence $\sigma_\Pi = \sigma_0, \ldots, \sigma_N$ with $\sigma_i \in \mathbb{F}_p^\times \times \{0,1\}^{n \cdot w}$ recursively as

$$\sigma_0 = \dot{\rho} \circ \pi_0(0^{w \cdot (n+1)}) \quad \left(= \rho\Big(\pi_0(0^{w \cdot (n+1)})^{(1)}\Big) \| \pi_0(0^{w \cdot (n+1)})^{(2 \ldots n+1)} \right), \text{and}$$

$$\forall i > 0 \ : \ \sigma_i = \dot{\rho} \circ \dot{\pi}_i(\sigma_{i-1}) \quad \left(= \rho\Big(\pi_i(\sigma_{i-1}^{(1 \ldots \tilde{i})})^{(1)}\Big) \| \pi_i(\sigma_{i-1}^{(1 \ldots \tilde{i})})^{(2 \ldots \tilde{i})} \| \sigma_{i-1}^{(\tilde{i}+1 \ldots n+1)} \right).$$

See Figure 3.4 for an illustration of the computation of σ_Π. Thus, using σ_Π in our protocol results in a PoSW that is secure in the random permutation model, almost as efficient as the construction from [CP18], and at the same time achieves verification

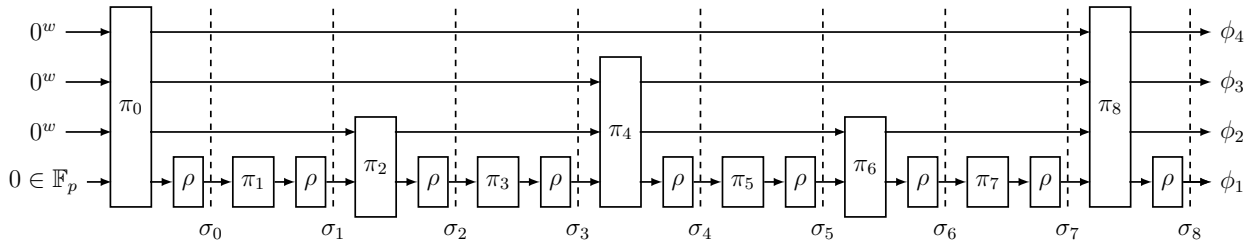

Figure 3.4: Illustration of the computation of $\sigma_\Pi = (\sigma_0, \ldots, \sigma_N)$ with $n = 3, N = 2^n = 8$.

of correctness as efficient as in sloth. When applied to a blockchain, our new PoSW allows extremely efficient rejection of wrong proofs while additionally providing sloth-like verification of correctness, which can be used whenever two or more distinct proofs pass the verification.

3.5 Open Problems

In this work we constructed a new PoSW in the random permutation model where verifying sequential work is as efficient as in [CP18] and verifying (the stronger property of) correctness is as efficient as in [LW17].

For the application of PoSW in cryptocurrencies, the efficiency of verification of both sequentiality as well as correctness of the computation are of concern. As for sequentiality, our construction offers an exponential gap between the work of the prover and that of the verifier. However, verifying correctness is only $\log p$ (say ≈ 1000) faster than recomputing the proof. Although this constant speedup may be sufficient for some applications, it may not be for others.

Therefore, the natural open problem is to achieve exponential gaps for both sequentiality and correctness at the same time while also maintaining practicality – the generic solutions using arguments of knowledge for general sequential functions are only of theoretical interest [MMV13].

Subsequent to this work, and indeed motivated by the application of PoSW to cryptocurrencies, a new primitive dubbed Verifiable Delay Functions (VDF), which is a PoSW with unique solutions, was defined and instantiated [BBBF18, Wes18, Pie18] – the practical instantiotions among these are based on algebraic assumptions. This effectively solves the open problem we pose here, yet it opens up a new more ambitious one, namely that of constructing practical VDFs that are quantum-secure. It would also be interesting to construct VDFs in the random permutation/oracle model or prove such a (black-box) construction is impossible.

Bibliography

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt BÃijnz, and Ben Fisch. Verifiable delay functions. Cryptology ePrint Archive, Report 2018/601, 2018. `https://eprint.iacr.org/2018/601`.

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

[chi18] Chia: Green money for a digital world. `https://chia.net/`, 2018. [Online; accessed 19-June-2018].

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 451–467, 2018.

[CPS08] Jean-Sébastien Coron, Jacques Patarin, and Yannick Seurin. The random oracle model and the ideal cipher model are equivalent. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 1–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

[DGK17] Yevgeniy Dodis, Siyao Guo, and Jonathan Katz. Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 473–495, Cham, 2017. Springer International Publishing.

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

[DSKT16] Dana Dachman-Soled, Jonathan Katz, and Aishwarya Thiruvengadam. 10-round feistel is indifferentiable from an ideal cipher. In Marc Fischlin and

Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 649–678, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[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.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

[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.

[HKT11] Thomas Holenstein, Robin Künzler, and Stefano Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 89–98, New York, NY, USA, 2011. ACM.

[LW17] Arjen K. Lenstra and Benjamin Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *IJACT*, 3(4):330–343, 2017.

[May93] Timothy C. May. Timed-release crypto. *http://www.hks.net/cpunks/ cpunks-0/1460.html*, 1993.

[MMV11] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 39–50. Springer Berlin Heidelberg, 2011.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 373–388, New York, NY, USA, 2013. ACM.

[Nak08] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008. [Online; accessed 19-June-2018].

[Pie18] Krzysztof Pietrzak. Simple verifiable delay functions. Cryptology ePrint Archive, Report 2018/627, 2018. https://eprint.iacr.org/2018/627.

[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.

[RSW00] Ronald L. Rivest, Adi Shamir, and David Wagner. Time-lock puzzles and timed-release crypto. *Technical Report MIT/LCS/TR-684, MIT*, February 2000.

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

[Wes18] Benjamin Wesolowski. Efficient verifiable delay functions. Cryptology ePrint Archive, Report 2018/623, 2018. https://eprint.iacr.org/2018/623.

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