Derandomization from Time-Space Tradeoffs

Oliver Korten
Columbia University, New York, NY, USA

Abstract

A recurring challenge in the theory of pseudorandomness and circuit complexity is the explicit construction of "incompressible strings," i.e. finite objects which lack a specific type of structure or simplicity. In most cases, there is an associated **NP** search problem which we call the "compression problem," where we are given a candidate object and must either find a compressed/structured representation of it or determine that none exist. For a particular notion of compressibility, a natural question is whether an efficient algorithm for the compression problem would aide us in the construction of incompressible objects. Consider the following two instances of this question:

1. Does an efficient algorithm for circuit minimization imply efficient constructions of hard truth tables?
2. Does an efficient algorithm for factoring integers imply efficient constructions of large prime numbers?

In this work, we connect these kinds of questions to the long-standing challenge of proving time-space tradeoffs for Turing machines, and proving stronger separations between the RAM and 1-tape computation models. In particular, one of our main theorems shows that modest time-space tradeoffs for deterministic exponential time, or separations between basic Turing machine memory models, would imply a positive answer to both (1) and (2). These results apply to the derandomization of a wider class of explicit construction problems, where we have some efficient compression scheme that encodes n-bit strings using $< n$ bits, and we aim to construct an n-bit string which cannot be recovered from its encoding.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic

Keywords and phrases Pseudorandomness, circuit complexity, total functions

Digital Object Identifier 10.4230/LIPIcs.CCC.2022.37

Related Version *Full Version*: `https://eccc.weizmann.ac.il/report/2022/025/`

Funding This research is supported by NSF Grant CCF-1763970.

Acknowledgements The author would like to thank Christos Papadimitriou and Mihalis Yannakakis for their support and guidance throughout the completion of this work.

1 Introduction

Mathematicians have long been familiar with the curious phenomenon of a *non-constructive proof*: an argument which demonstrates the existence of an object satisfying some special property, but which fails to indicate a particular example of such an object. The advent of complexity theory has provided us with a formal way of defining the level of "inherent constructivity" in a theorem: we can say that an existence theorem is constructive if there is an accompanying polynomial time algorithm supplying an example of one of the objects the theorem proves to exist, and is inherently non-constructive if no such algorithm exists. Papadimitriou initiated a formal complexity-theoretic treatment of this topic three decades ago [18], where he provided a taxonomy of total search problems (problems that always have solutions) in **NP** by classifying them based on the strength of the lemma guaranteeing the existence of a solution on all instances.

When a particular theorem appears to be inherently non-constructive, in that no polynomial time algorithm can witness the solutions it guarantees, one perspective we can take is that this theorem is "effectively false" in a certain context, despite being irrefutably true in general. This is essentially the outlook presented by Yao in his seminal paper introducing the basis of theoretical cryptography [24]. Here, Yao focuses on the central tenets of Shannon's information theory. He argues that although Shannon's theorems are of course provably correct, in many scenarios it appears computationally infeasible to witness their truth. For example, an output of some function $f\colon \{0,1\}^n \to \{0,1\}^{2n}$ under the uniform distribution on $\{0,1\}^n$ has entropy n, and thus by a result of Shannon can be encoded on average using n bits. However for an observer who sees only the outputs of this distribution, there seems to be no efficient way to generate such a code without the ability to efficiently invert f. If we posit that there are some specific functions f for which it is in fact impossible to efficiently realize Shannon's theorem in this sense, one might venture to say that Shannon's theorem *effectively fails* for f in the computational realm, perhaps allowing us to carry out tasks which would, in the absence of computational constraints, be deemed impossible. Indeed it is widely conjectured that there are efficiently computable functions f of this form, and this conjecture underpins the security of many cryptographic protocols. Similar situations are abundant in the field of cryptography, where the computational infeasibility of *witnessing* impossibility theorems from information theory allows us to effectively bypass them.

With this perspective in mind, let us now turn our attention to a special family of non-constructive proofs of great interest to complexity theorists, proofs which guarantee the existence of "pseudorandom objects." Key examples include the existence of truth tables of high circuit complexity and of pseudorandom generators capable of derandomizing algorithms. The task of making these proofs constructive is often referred to as an "explicit construction problem," since the goal is to print one explicit example of an object possessing some pseudorandom property. In [14] it is shown that a broad collection of such problems can be reduced to the following more general task: given some efficiently computable function $f\colon \{0,1\}^{n-1} \to \{0,1\}^n$, find an n-bit string outside the range of f. The existence of a solution is guaranteed by the "dual weak pigeonhole principle," and indeed this principle guarantees that a *randomly chosen* string is a solution with high probability. The question at hand is whether this principle is *inherently nonconstructive*. Unlike the case of Shannon's theorems on information transmission, the prevailing wisdom in complexity theory is this theorem *can be made constructive*: it is widely conjectured that exponential time requires exponential circuit size, and by [14], this would in fact imply a generic non-trivial[1] "witnessing" algorithm for the dual weak pigeonhole principle, i.e. an algorithm which produces n-bit strings outside the range of any such f.

In search of evidence for this widely-conjectured belief that the weak pigeonhole principle can be made more constructive, the starting point of this work is to imagine what the computational landscape would look like if it were false, i.e. if we lived in a world where there was an "effective counterexample" to the weak pigeonhole principle. In particular, let us model this scenario by supposing that there are a pair of efficiently computable functions $\mathcal{G} = \{g_n\colon \{0,1\}^n \to \{0,1\}^{n-1}\}_{n\in\mathbb{N}}$, $\mathcal{F} = \{f_n\colon \{0,1\}^{n-1} \to \{0,1\}^n\}_{n\in\mathbb{N}}$, such that no polynomial time algorithm is able to construct an n-bit string x such that $f_n(g_n(x)) \neq x$ in $poly(n)$ time (for more then finitely many n). Note that this is a slightly different version of the weak pigeonhole principle then what we defined in the previous paragraph: here we are given both the length-increasing function f and a supposed inverse g, and the pigeonhole

[1] The implied witnessing algorithm runs in $\mathbf{P}^{\mathbf{NP}}$, whereas the trivial upper bound for this problem is $\mathbf{\Sigma_2^P}$.

principle tells us that $f \circ g$ cannot be the identity. In this hypothetical world where $\mathcal{F}, \mathcal{G}$ are an "effective counterexample" to the weak pigeonhole principle, we have access to an efficient compression scheme which allows us to encode any n-bit string using $n-1$ bits. What improbable feats can be accomplished in such a world?

In this work we give one answer to this question: if the weak pigeonhole principle *effectively fails* in the above sense, we can utilize this failure to construct an efficient data structure which allows us to simulate RAM computations in low space and near-linear time on a 1-tape Turing machine. Stated in contrapositive, mild time-space tradeoffs for simulating RAM machines on a 1-tape Turing machine would imply a generic algorithm to witness the weak pigeonhole principle, and in turn would have significant consequences in the theory of pseudorandomness and circuit complexity.

1.1 Our Contributions

1.1.1 Derandomization from Time-Space Tradeoffs

Let $\mathcal{C}, \mathcal{D} = \{C_n \colon \{0,1\}^n \to \{0,1\}^{n-1}\}_{n\in\mathbb{N}}, \{D_n \colon \{0,1\}^{n-1} \to \{0,1\}^n\}_{n\in\mathbb{N}}$. We call such a pair a "uniform compression scheme," where $\mathcal{C}$ is the "compressor" which encodes n-bit "messages" by $n-1$-bit "codewords," and $\mathcal{D}$ is the "decompressor" which maps $n-1$-bit codewords to the messages they represent[2]. We use the term "uniform" since in the relevant cases, $\mathcal{C}$ and $\mathcal{D}$ will each be described by a single Turing machine which computes C_n (resp. D_n) for all n. We will say that a string $x \in \{0,1\}^n$ is "incompressible" for such a scheme if $D_n(C_n(x)) \neq x$, i.e. if $\mathcal{D}$ cannot be used to recover the message x from the codeword assigned to it by $\mathcal{C}$. In this paper, we study the complexity of the *explicit construction task* of generating incompressible strings for uniform compression schemes. This task can be viewed naturally as a derandomization problem, since a randomly chosen n-bit string will be incompressible with high probability with respect to any fixed compression scheme.

Our results then show how to derive efficient derandomized algorithms for generating incompressible strings, assuming certain *uniform* lower bounds. The lower bounds we consider are time-space tradeoffs for simulating RAM by 1-tape machines. Roughly, they posit that there are problems solvable in time T on a RAM machine which cannot be solved on a 1-tape machine using $T^{0.01}$ space and $T^{1.01}$ time. Depending on certain features of the compression scheme and the resources available to the explicit construction algorithm, the specifics of the tradeoff assumption will vary, as we shall explain in Table 1.

Initially, we consider the case of "poly-time compression schemes", where both $\mathcal{C}$ and $\mathcal{D}$ are computable in polynomial time. We show that for such schemes, incompressible strings can be constructed deterministically in polynomial time assuming time-space tradeoffs for deterministic exponential time:

▶ **Theorem** (Theorem 31). *Suppose there is some exponential time bound $T(n) \geq 2^{\Omega(n)}$ and some $\epsilon > 0$ such that it is impossible to simulate $T(n)$-time RAM computations on 1-tape Turing machines that use $T(n)^{1+\epsilon}$ time and $T(n)^{\epsilon}$ space. Then for any poly-time compression scheme, there is a polynomial time algorithm that produces incompressible n-bit strings for this scheme on input 1^n (for infinitely many n).*

Constructing incompressible strings for poly-time compression schemes is a natural and quite broad derandomization problem. Theorem 31 gives a novel connection between derandomizing this task and proving *uniform* time-space tradeoffs– in contrast, the assumptions

[2] The formal definition given in Section 3 is slightly more general, but we will focus on this special case for now.

previously required to derandomize this sort of problem require a lower bound against *non-uniform* algorithms, such as circuits. To put this derandomization task in a more familiar context, in Section 3.2 we show that several well-studied explicit construction problems, such as the construction of large prime numbers or the construction of truth tables of high circuit complexity/formula size, can be reduced to the problem of finding incompressible strings for some uniform compression scheme. In each case, the compression scheme will either be computable efficiently, or efficiently with access to an oracle for some search problem in **FNP** which is not known to be **NP**-complete.

To state our other main results in their most interesting form, we focus our attention now on the construction of strings/truth tables of high complexity with respect to some non-uniform complexity measure, such as formula size, circuit size, or time-bounded Kolmogorov complexity. In each case there is an associated **NP** search problem, where we are given a string/truth table and must find a small formula/circuit/program computing it (if one exists); we call this the "compression problem." The following table shows how our three main theorems relate various time-space tradeoff hypotheses to such problems:

Table 1 Main Results. We use the shorthand $T = T(n)$.

Hypothesis: For some exponential time bound T and some $\epsilon > 0$...	**Implication** :
$\mathbf{RAM\text{-}TIME}[T] \not\subseteq \mathbf{1\text{-}TISP}[T^{1+\epsilon}, T^{\epsilon}]$	If the compression problem has a polynomial time algorithm, then incompressible strings can be constructed in polynomial time.
$\mathbf{NTIME}[T] \not\subseteq \mathbf{1\text{-}NTISPG}[T^{1+\epsilon}, T^{\epsilon}, T^{\epsilon}]$	Incompressible strings can be constructed in polynomial time with an **NP**-oracle. In particular, $\mathbf{E}^{\mathbf{NP}} \not\subseteq \mathbf{size}[2^n/2n]$.
$\mathbf{RAM\text{-}TIME}[T] \not\subseteq \mathbf{1\text{-}NTISPG}[T^{1+\epsilon}, T^{\epsilon}, T^{\epsilon}]$	Incompressible strings can be constructed in polynomial time with an oracle for the compression problem.

The classes seen on the left hand side will be defined formally in Section 2.2, but we give a brief explanation here so that the table can be interpreted appropriately. $\mathbf{TIME}[T(n)]$ and $\mathbf{NTIME}[T(n)]$ are defined in the standard way using multitape machines; the prefixes **1** and **RAM** indicate, respectively, either a restriction of the model to 1-tape machines or a strengthening to random access machines. The class $\mathbf{TISP}[T(n), S(n)]$ consists of problems decidable simultaneously in time $T(n)$ and space $S(n)$. Finally, $\mathbf{NTISPG}[T(n), S(n), G(n)]$ consists of problems decidable by a nondeterministic machine running in time $T(n)$ which, on every computation path, uses at most $S(n)$ space and $G(n)$ nondeterministic guesses.

For a concrete example of how to apply these results, lets focus on the compression problem Circuit Synthesis (given a truth table find a small circuit for it if one exists), and the explicit construction problem of producing truth tables of exponential circuit complexity. The first hypothesis in the above table implies that if Circuit Syntheses is in **P** then there is a polynomial time construction of hard truth tables (i.e. **E** has exponential circuit complexity). The second hypothesis implies unconditionally that there is an efficient **NP** oracle construction of hard truth tables (i.e. $\mathbf{E}^{\mathbf{NP}}$ has exponential circuit complexity). Finally, the third and strongest hypothesis tells us that there is a polynomial time construction of hard truth tables using an oracle for Circuit Synthesis.

The case of large prime construction does not fit in as neatly to the above picture, as the scheme we devise for this problem will require a factoring oracle for both the compressor and decompressor (while the above problems have a polynomial-time computable decompressor). In this case we get the following result:

▶ **Theorem** (Corollary 35). *Under the hypothesis in the top row of the above table, a polynomial time algorithm for factoring implies a polynomial time algorithm to construct* $32n$*-bit primes of magnitude* $> 2^n$ *for infinitely many* n*.*

If we forgo the questionable assumption that factoring lies in **P**, we get:

▶ **Theorem** (Corollary 36). *One of the following is true:*

1. *For every exponential time bound* T *and every* $\epsilon > 0$*, every language decidable in time* $T(n)$ *on a RAM machine can be decided in time* $T(n)^{1+\epsilon}$ *and space* $T(n)^\epsilon$ *by a 1-tape machine with a factoring oracle, which makes oracle calls of length at most* $T(n)^\epsilon$*.*
2. *There is a polynomial time algorithm with a factoring oracle that generates* $32n$*-bit primes of magnitude* $> 2^n$ *for infinitely many* n*.*

The problem of deterministically generating large primes has been investigated previously in several works, and was notably the subject of the Polymath 4 project [20]. In the public discussion forums for this project, it was explicitly asked whether a polyomial time algorithm for factoring, or more generally an oracle for factoring, would help. More recently, Oliviera and Santhanam gave a subexponential time "pseudodeterministic" construction of large primes [17], using only the fact that primality is testable in **P** [1] and that primes occur with non-negligible frequency.

1.1.2 BPP and the Weak Pigeonhole Principle

In Section 6 we briefly consider the relationship between various search problems associated with the weak pigeonhole principle, and the "full derandomization task" characterized by the class **prBPP**. Observe that the problem introduced above of finding incompressibe strings for uniform compression schemes can be generalized to a **TFNP** search problem – instead of considering compression schemes generated by uniform Turing machines, we can consider the search problem where a compression scheme of some fixed length is given as input in the form of a pair of boolean circuits:

▶ **Definition 1.** *In* Lossy Code*, we are given as input a pair of circuits* $C\colon \{0,1\}^n \to \{0,1\}^{n-1}$*,* $D\colon \{0,1\}^{n-1} \to \{0,1\}^n$*, and must output some* $x \in \{0,1\}^n$ *such that* $D(C(x)) \neq x$*.*

While finding a deterministic algorithm for Lossy Code appears to be a quite generic derandomization problem, it would be a major breakthrough to show that Lossy Code captures the "full derandomization problem:" since Lossy Code lies in **TFNP**, if **prBPP** reduces to Lossy Code then **BPP** $\subseteq$ **NP**, which is a notorious open problem. In Section 6, we show that a natural generalization of Lossy Code, where we allow the compressor C to be randomized, is indeed strong enough to characterize **prBPP** precisely. The formal problem is as follows:

▶ **Definition 2.** *In* R-Lossy Code*, we are given as input circuits* $C\colon \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^{n-1}$*,* $D\colon \{0,1\}^{n-1} \to \{0,1\}^n$*, and must find some* x *such that* $Pr_r[D(C(x,r)) = x] < \frac{1}{2}$*.*

In this work we demonstrate:

▶ **Theorem** (Theorem 40). R-Lossy Code *is complete for* **prBPP** *under deterministic Turing reductions.*

The relation between pigeonhole principle search problems and more standard derandomization problems is summarized in Figure 1.

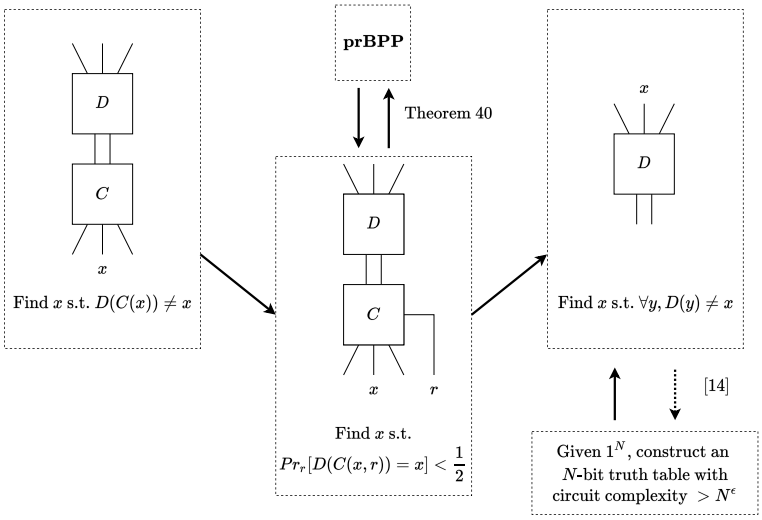

Figure 1 Relations between the total search problems associated with various weak pigeonhole principles, and standard derandomization problems. Solid arrows represent deterministic polynomial time reductions, while dotted arrows represent **NP**-oracle reductions.

The proof of Theorem 40 is mostly standard, following quite directly from Yao's next bit predictor lemma. From the proof of this theorem we are able to extract the following interesting corollary:

▶ **Corollary** (Corollary 41, Informal). *If the fixing of the leftover bits in Yao's hybrid argument can be derandomized, then* **BPP** $\subseteq$ **NP**. *Indeed, under this assumption, every problem in* **prBPP** *reduces to the search problem* Lossy Code *in* **PPP** $\subseteq$ **TFNP**.

In other words, derandomizing a particular step in Yao's classical argument implies a quite universal derandomization of **prBPP**.

1.2 Relation to Prior Work

Hardness vs. Randomness

As mentioned above, the task of constructing incompressible strings can be naturally viewed as a derandomization problem, and is thus amenable to the standard hardness/randomness paradigm. In particular, when the compression scheme is polynomial time computable as in Theorem 31, the standard hardness assumptions used to derandomize **BPP** (e.g. [8]) would suffice to yield a polynomial time construction of incompressible strings. Such hardness assumptions require a lower bound for a language in **E** against some non-uniform model of computation, such as boolean circuits. In contrast, Theorem 31 gives a hardness/randomness connection for this problem which only requires a *uniform lower bound* for a a language in **E**, in particular a lower bound against low-space algorithms running in slightly more time using a weaker memory model.

The Easy Witness Lemma

Perhaps the closest prior work to our results is the "Easy Witness Method," initiated by [11] and furthered in [7] and [23], which roughly says that assuming $\mathbf{E}^{\mathbf{NP}}$ has small circuits, any nondeterministic exponential time computation must have witnesses of low circuit complexity. This immediately implies that unless $\mathbf{E}^{\mathbf{NP}}$ requires large circuits, we can efficiently simulate $\mathbf{NTIME}[2^n]$ using limited nondeterminism by "guessing a small circuit." In the full version, we show that this old method (along with one other well-known tool) is in fact enough to prove the second implication in Table 1, although this connection to time-space tradeoffs does not seem to have been noted previously. However, this proof heavily utilizes the distinctive power of nondeterminism, whereby additional "guesses" allow us to vastly simplify the verification procedure, and does not seem to extend to our other two main results.

1.3 Known Time-Space tradeoffs

Finally, we cover the known results on time-space tradeoffs and separations between the RAM and 1-tape models. We first emphasize the following: all three time-space tradeoff hypotheses stated on the left-hand side of Table 1 are known to hold *unconditionally* when the time bound T is $O(n)$. In particular, an old result of Maass [15] shows that the set of palindromes is recognizable in quasilinear time on a deterministic RAM machine, but requires $\Omega(n^2)$ time on a nondeterministic 1-tape machine. However, Maass's proof is essentially a counting argument which crucially relies on the entropy of the input being comparable in magnitude to the total computation time, which is no longer the case for exponential time computations. When it comes to separating the RAM and 1-tape models for generic time bounds, a result of [21] shows that there must be *some* slowdown when simulating a RAM on a 1-tape machine for *any time bound* greater then $n \log \log n$, but the slow-down is an astronomically small multiplicative factor, on the order of $\log \log T(n)$.

Another sequence of works ([2], [3], [22] among others) has shown unconditionally that nondeterministic linear time cannot be solved in $\mathbf{TISP}[n^{1+\epsilon}, n^{\epsilon}]$ for certain fixed values of $\epsilon > 0$, even when the simulating machine is given access to RAM. These results can be scaled to larger time bounds as follows: for any time-constructible T, $\mathbf{\Sigma_2 TIME}[T(n)]$ is not contained in $\mathbf{TISP}[T(n)^{1+\epsilon}, T(n)^{\epsilon}]$ for certain fixed values of $\epsilon > 0$. Such results are obtained by combining hierarchy theorems with a fast simulation result, whereby a $\mathbf{\Sigma_2}$ machine can simulate a $\mathbf{TISP}[T(n)^{1+\epsilon}, T(n)^{\epsilon}]$ computation in time $T(n)^{1-\delta}$ time for some fixed $\epsilon, \delta > 0$. It is evident that such results will not be sufficient for our purposes, as they only rule out efficient low-space simulations of $\mathbf{\Sigma_2}$ computations by deterministic machines.

1.4 Main Tool: the J-tree

The basis of our main proofs is a construction which we call the "J-tree." Roughly speaking, the J-tree is a data structure which allows us to store an array of T elements, subject to fast update/query operations, using significantly less then T bits. While this task is information-theoretically impossible (by the weak pigeonhole principle), the J-tree uses a compression scheme C, D to perform its operations, and whenever it fails in its role as a data structure, it is able to print out an incompressible string for this scheme. Thus, assuming no polynomial time algorithm can witness the weak pigeonhole principle for C, D, no polynomial time algorithm can witness the failure of this data structure. This will allow our low-space simulations to operate a RAM memory using low space on a 1-tape machine, in such a way that if the simulation fails, then a polynomial time algorithm can witness this failure, and thus witness the weak pigeonhole principle for C, D. The core construction underlying the

J-tree has a long and intriguing history, occurring in some form or another in [16, 5, 19, 9], and [14]. The details of this history and its relation to our results are explained in the full version; our main novel contribution to this line of work is the "J-tree Update Lemma" (Lemma 25).

2 Preliminaries

2.1 Basics

Our notation for basic concepts is standard, e.g. all logarithms are base 2, we use $[n]$ to denote $\{1, \ldots, n\}$, $|x|$ to denote length of a binary string. We will define the following precise notion of an "exponential time bound:"

▶ **Definition 3.** *We say a function $T\colon \mathbb{N} \to \mathbb{N}$ is an "exponential time bound" if T is time constructible, and there exist constants $0 < \beta < B$ such that for sufficiently large n, $2^{\beta n} \leq T(n) \leq 2^{Bn}$.*

This is in contrast to the more general notion of exponential growth where we have $2^{n^\beta} \leq T(n) \leq 2^{n^B}$.

2.2 Machine Models

We start by precisely defining the various machine models and complexity classes that will be used. We begin with a definition of "random access memory" or "RAM" machines:

▶ **Definition 4** (RAM machines)**.** *A RAM machine is a turing machine equipped with two binary tapes, one called the "auxilliary tape" and the other called the "addressing tape." We collectively refer to these as the "linear tapes." Both linear tapes admit the same operations as a standard Turing machine tape, where in one step we can move the head left/right and read or modify a cell of the tape. In addition, there is an associated "RAM memory" consisting of a sequence of binary variables $A_1, A_2, \ldots$. The addressing tape has two additional operations that allow it to interact with the RAM memory. There is an* Update *operation, which in one step sets $A_i = b$, where i is the integer whose binary representation lies to the left of the addressing tapehead, and b is the bit currently read by the auxiliary tape. There is also a* Load *operation, which in one step sets the cell pointed to by the auxiliary tapehead to the value A_i, where again i is the integer whose binary representation lies to the left of the addressing tapehead. At the start of the computation on input x, the RAM cells $A_1, \ldots, A_{|x|}$ are initialized to the bits of x in order, and $A_{|x|+1}, \ldots$ are initialized to 0. The addressing tape is then initialized to $|x|$, so that the machine knows where the input ends.*

We will use the following simplification lemma for RAM computations later, which follows easily from the use of balanced binary search trees:

▶ **Lemma 5** ([6])**.** *A RAM machine running in time $T(n)$ can be simulated in time $T'(n) = \tilde{O}(T(n))$ by a RAM machine that uses $O(\log T'(n))$ space on its addressing and auxiliary tapes, and uses only its first $T'(n)$ RAM cells.*

We will also make reference to k-tape Turing machines, for which we omit a formal definition as this model is standard. However, we will at some points concern ourselves with multi-tape machines equipped with oracles for languages/functions, and here we will need to be precise about the oracle access mechanism:

▶ **Definition 6** (Oracle Turing Machines). *For a function $F\colon \{0,1\}^* \to \{0,1\}^*$, an F-oracle k-tape Turing machine has k standard read/write "work tapes," in addition to an oracle tape where in one step the machine can replace the leading cells of the oracle tape with $F(x)$, where x is the string lying to the left of the oracle tapehead prior to the oracle call; after this the oracle tapehead is moved to the first cell of the oracle tape. In some cases we will give a machine access to several oracles $F_1, F_2, \ldots, F_k$ (for a fixed constant k), in which case each gets its own oracle tape. The oracle tape(s) and the first work tape share a single tapehead, and in one move this tapehead can move from the first cell of the work tape to the first cell of one of the oracle tapes, or vice-versa.*

The space usage of an oracle k-tape machine is defined to be the sum of the space used on all of its work tapes and oracle tapes.

The sharing of a single tapehead between the oracle tapes and the first work tape is only relevant for 1-tape machines, where we don't want the machine to "cheat" and use it's oracle tapes as additional work tapes.

▶ **Definition 7** (Time Bounded Classes). *For a function $T\colon \mathbb{N} \to \mathbb{N}$, let $\mathbf{TIME}[T(n)]$ denote the class of languages which are decidable by a deterministic multi-tape Turing machine running in time $O(T(n))$. Let $\mathbf{RAM\text{-}TIME}[T(n)]$ be defined analogously for RAM machines.*

▶ **Definition 8** (Time-Space Classes). *For functions $T, S\colon \mathbb{N} \to \mathbb{N}$, let $\mathbf{TISP}[T(n), S(n)]$ denote the class of languages which are decidable by a deterministic multi-tape Turing machine running simultaneously in time $O(T(n))$ and total space $O(S(n))$. Let $\mathbf{1\text{-}TISP}[T(n), S(n)]$ be defined identically, but with the additional restriction that the machine uses only one tape.*

Finally, we define an analogous "time-space" class for nondeterministic time:

▶ **Definition 9** (Nondeterministic Time-Space Classes). *For functions $T, S, G\colon \mathbb{N} \to \mathbb{N}$, let $\mathbf{NTISPG}[T(n), S(n), G(n)]$ ("nondeterministic time, space, guess") denote the class of languages decidable by a nondeterministic multi-tape Turing machine such that on any computation path on an input of length n, the machine spends time $O(T(n))$, uses space at most $O(S(n))$, and makes at most $O(G(n))$ non-deterministic guesses. Let $\mathbf{1\text{-}NTISPG}[T(n), S(n), G(n)]$ be defined identically, but with the additional restriction that the machine uses only one tape. We define $\mathbf{NTIME}[T(n)]$ in the standard way, which is analogous to the above but with no restriction on space usage or nondeterminism.*

We make note of the following result, which tells us that for nondeterministic computations the multi-tape and RAM models are roughly equivalent:

▶ **Lemma 10** ([6]). *Any time-$T(n)$ computation on a nondeterministic RAM machine can be simulated in $\tilde{O}(T(n))$ time on a nondeterministic multi-tape machine.*

We will thus not bother to explicitly define a nondeterministic RAM model, though the definition for the deterministic case naturally extends.

3 Pigeonhole Principles and Compression Schemes

We now define the basic formalization of the weak pigeonhole principle we will investigate in this work, and introduce the relevant terminology.

3.1 Compressors and Decompressors

▶ **Definition 11.** *Let $D\colon \{0,1\}^m \to \{0,1\}^n$, with $m < n$. We call such a map which extends its input length a "decompresser." The "code length" of this decompressor is m, and its "message length" is n.*

The terminology should be interpreted as follows: for a string x such that $D(y) = x$, y functions as an m-bit compressed representation (or "codeword") for the n-bit "message" x, and D functions as an an algorithm which lets us "decompress" this codeword into the message it represents.

By the *dual weak pigeonhole principle*, if $n > m$, any function mapping 2^m "pigeons" to 2^n "holes" must leave some hole empty. We thus know there must exist an $x \in \{0,1\}^n$ such that $\forall y \in \{0,1\}^m$, $D(y) \neq x$. We call such a string x an "empty pigeonhole" for the decompressor D.
The primary subject of [14] was the problem EMPTY, originally introduced in [13], where we are given a decompressor specified by a boolean circuit and must output one of its empty pigeonholes. In this work, we will study a slight modification of this problem, where we are given both a compressor and a decompressor, and must and must find a witness to the fact that some message is not recoverable from its codeword. This was referred to by Jeřábek as the "retractive pigeonhole principle."[10].

▶ **Definition 12.** *Let $C\colon \{0,1\}^n \to \{0,1\}^m, D\colon \{0,1\}^m \to \{0,1\}^n$, where $m < n$; we call such a C a "compressor", and collectively we will refer to C, D as a "compression scheme." Again, m and n are the "code length" and "message length" respectively. By the pigeonhole principle, we know that there must exist some $x \in \{0,1\}^n$ such that $D(C(x)) \neq x$; we call such an x "incompressible" with respect to the scheme C, D.*

Note that when $x \in \{0,1\}^n$ is an empty pigeonhole for a decompressor D, it is necessarily incompressible for *all schemes* C, D which use D as the decompressor. The converse is not true in general, but when it is we use the following terminology:

▶ **Definition 13.** *We call C a "proper compressor" for D if C, D satisfy the following for all $x \in \{0,1\}^n$: if there exists an $y \in \{0,1\}^m$ such that $D(y) = x$, then $D(C(x)) = x$.*

By definition, when C is a proper compressor for D, the incompressible strings for the scheme C, D correspond exactly to the empty pigeonholes for D. It should be noted that when D is polynomial time computable, or is specified by some boolean circuit, there always exists a proper compressor for D computable efficiently with an **NP** oracle, which simply finds the lexicographically first preimage of a string under D or else outputs something arbitrary when none exist. In this sense, the work of [14], which studies the complexity of EMPTY with respect to **NP**-oracle reductions, was essentially studying a special case of this problem, where the compressor is the "canonical proper compressor" which searches for the lexicographically first preimage.

For most of this work it will be convenient to focus on the special case of functions which exactly double their input length:

▶ **Definition 14.** *When a compression scheme has code length n and message length $2n$, we will say that it has "stretch 2."*

We will utilize the following lemma, which tells us that if our goal is to find an incompressible string with respect to some scheme, we can assume it has stretch 2 without loss of generality:

▶ **Lemma 15.** *Let C, D be a compression scheme with code length n and message length m. Then there is another scheme C', D' of code length n and message length $2n$, such that C' is computable in poly(m) time with an oracle for C, D' is computable in poly(m) time with an oracle for D, and such that given an incompressible string for C', D', we can construct an incompressible string for C, D in poly(m) time with oracles for C, D.*

Proof. This is proven in [14] for the special case when C is a proper compressor for D, but the exact same proof extends to the more general case stated here. ◀

In [14], the problem EMPTY was studied as a search problem, where the decompressor D is provided as input (in the form of a circuit), and the goal is to find one of its empty pigeonholes. We could equivalently define such a search problem for the retractive pigeonhole principle, where we are given circuits computing C, D as input, and must output an incompressible string for this scheme (this search problem is considered in Section 6). However, for the purposes of our main results it will be more suitable to study a *uniform* version of this problem, where we have a sequence of schemes C_n, D_n of increasing size, and each are computable within some uniform complexity class.

▶ **Definition 16.** *A uniform compression scheme is a pair $\mathcal{C} = \{C_n \colon \{0,1\}^{\ell(n)} \to \{0,1\}^{m(n)}\}_{n\in\mathbb{N}}$, $\mathcal{D} = \{D_n \colon \{0,1\}^{m(n)} \to \{0,1\}^{\ell(n)}\}_{n\in\mathbb{N}}$ for some pair of time constructible functions $m, l \colon \mathbb{N} \to \mathbb{N}$ such that $m(n) < \ell(n)$ for all n, and $\ell(n)$ is bounded above by a polynomial in n.*

In this work we will concern ourselves with uniform compression schemes where $\mathcal{C}, \mathcal{D}$ are computable in polynomial time, each with access to different oracles, hence the name "uniform."

3.2 Particular Compression Schemes of Interest

We now introduce some uniform compression schemes whose incompressible strings have certain desirable properties for which no explicit constructions are currently known.

3.2.1 Compression Schemes for Non-Uniform Complexity Measures

We start with the case of hard truth tables. The following lemma is a well-known folklore result, for which a formal proof can be found in [14]:

▶ **Lemma 17.** *For sufficiently large $N \in \mathbb{N}$, there is function $\mathsf{f}_N \colon \{0,1\}^{N-1} \to \{0,1\}^N$ computable uniformly in $\tilde{O}(N^2)$ time such that if $x \in \{0,1\}^N$ has circuits of size at most $\frac{N}{2\log N}$, then x is in the range of f_N.*

We now define the search problem associated with finding preimages of strings under f_N, which has commonly been referred to as the "circuit synthesis problem:"

▶ **Definition 18** (Circuit Synthesis)**.** *Given a string $x \in \{0,1\}^N$, output an $N-1$ bit description of a circuit C of size at most $\frac{N}{2\log N}$ on $\lceil \log N \rceil$ variables such that $C(i) = x_i$ for all $0 \leq i < N$ if such a circuit exists, or else determine that no such circuit exists.*

This problem is the search variant of the more well-studied "Minimum Circuit Size Problem" [12], which is not known to admit a search-to-decision reduction. By the same arguments underlying Lemma 17, given an instance $x \in \{0,1\}^N$ of Circuit Synthesis, any circuit of the stated size can be represented using at most $N-1$ bits via a standard encoding so this search problem is well defined. Further, requiring the output to be specified in such an encoding does not increase the complexity of the problem, since the standard encoding can be computed efficiently from any description of such a circuit.

By definition, we see that there is a proper compressor for f_N computable in polynomial time with a Circuit Synthesis oracle[3], and thus any incompressible string for this scheme is an N-bit string with high circuit complexity.

We similarly have the following:

▶ **Lemma 19.** *For any fixed polynomial p, there is a uniform compression scheme whose decompressor is computable in polynomial time, and whose compressor is computable with an oracle for $K^{p(n)}$* Minimization *(given $x \in \{0,1\}^n$ find a short program $y \in \{0,1\}^{n-2}$ that prints x in $p(n)$ steps if one exists). The incompressible strings for this scheme have $K^{p(n)}$ complexity $\geq n-1$.*

▶ **Lemma 20.** *There is a uniform compression scheme whose decompressor is computable in polynomial time, and whose compressor is computable with an oracle for* Formula Synthesis *(given a truth table find a short formula if one exists). The incompressible strings for this scheme are truth tables of exponential formula size.*

We provide these as basic examples without defining the problems too formally; more generally it can be verified that for most reasonable non-uniform measures of complexity (which are bounded above by K^{poly}), such a uniform scheme exists whose decompressor is computable efficiently, whose compressor is computable with access to the relevant "compression problem," and whose incompressible strings have high complexity with respect to this measure.

3.2.2 Large Primes

We next construct a compression scheme related to the large prime construction problem. This scheme is unlike the previous ones, in that both the compressor and decompressor have the same complexity: each will require an oracle for factoring.

▶ **Theorem 21.** *There is a compression scheme R, P with code length $n + \lceil \log n \rceil + 3$ and message length $n + \lceil \log n \rceil + 4$, such that R, P are each computable uniformly in polynomial time with a factoring oracle, and given an incompressible string for this scheme, a $32n$-bit prime of magnitude $> 2^n$ can be constructed in polynomial time with a factoring oracle.*

A proof of Theorem 21 can be found in the full version. This theorem is an algorithmic analogue of a well known result of Paris, Wilkie, and Woods [19], and our proof follows from analysing the computational resources needed to carry out their construction.

4 J-Trees

In this section we develop the core tool used in our main result, namely the "J-tree." The J-tree is, informally, a data structure "solving" the following information-theoretically impossible task: store an array of T elements, subject to efficient updates and queries, using significantly fewer then T bits. While such a data structure cannot exist unconditionally, the J-tree will take as input a compression scheme C, D, and will be set up so that when it fails in its capacity as a data structure, it prints out an incompressible string for C, D. In other words, if it is hard to witness the weak pigeonhole principle for the scheme C, D, then it is hard to find a sequence of updates/queries which causes our data structure to fail.

[3] There is some subtly when considering oracles for search problems like Circuit Synthesis which have multiple valid solutions on each input, see the full version for a discussion of how our results are robust to this potential issue.

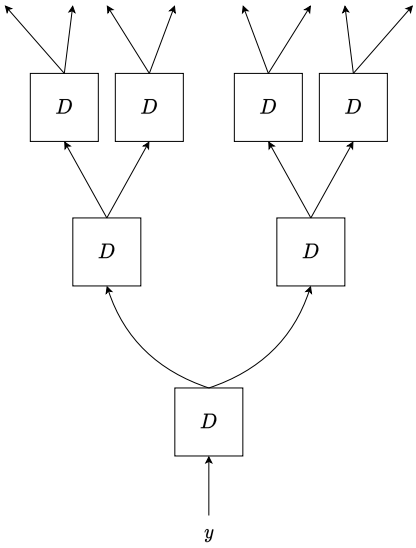

Figure 2 A J-tree of depth 3. Each arrow consists of n wires, where n is the code length of the decompressor D.

▶ **Definition 22** (J-trees). *Let $D\colon \{0,1\}^n \to \{0,1\}^{2n}$ be a decompressor of code length n and stretch 2. We define $D_0, D_1\colon \{0,1\}^n \to \{0,1\}^n$ to be the maps obtained by computing D and taking the first n and last n bits of output respectively. Now, for any binary string $x \in \{0,1\}^*$, we define $D_x\colon \{0,1\}^n \to \{0,1\}^n$ as follows. When $|x| = 0$, D_x is the identity, and when $|x| = 1$, D_0, D_1 are defined as above. In the general case, when $x = b_1 \cdots b_k$ for some $b_1, \ldots, b_k \in \{0,1\}$, D_x is defined as:*

$$D_x = D_{b_k} \circ \cdots \circ D_{b_1}$$

Now, for a particular input $y \in \{0,1\}^n$, we can define the function $D[y]\colon \{0,1\}^* \to \{0,1\}^n$ as follows. For each $x \in \{0,1\}^*$:

$$D[y](x) = D_x(y)$$

For an integer k, we then define $D^k[y]\colon \{0,1\}^k \to \{0,1\}^n$, which is simply the restriction of $D[y]$ to the domain $\{0,1\}^k$.

We will refer to $D^k[y]$ as a "J-tree," D as its decompressor, y as its "seed," and k as its "depth."

It will be useful to visualize a J-tree as a binary tree (hence the name). Refer to Figure 2 which illustrates a J-tree $D^k[y]$ where $k = 3$. For a given D and depth k, we can construct a tree-like circuit which starts with one copy of D, then feeds each of its two n-bit output blocks into another copy of D, and so on for k iterations, ultimately yielding a circuit with n inputs and $2^k n$ outputs which has the structure of a perfect binary tree of depth k. If we now fix the inputs to some value y (the seed), by passing y through this tree-like circuit we obtain 2^k n-bit values at the output. As seen in the figure, each of the 2^k n-bit values is associated uniquely with a leaf in this binary tree of depth k, and thus can be specified by a k-bit index indicating whether to move left or right at each step along a root-to-leaf path. $D^k[y]$ is then the function which, given the description of such a path, returns the value at the corresponding leaf.

We will think of a J-tree $D^k[y]$ operationally as a data structure which stores an array of 2^k n-bit values, one for each for each of its 2^k "leaves", where the i^{th} value is simply $D^k[y](i)$. The state of the data structure is described purely by y and D, which in our use cases will require far fewer bits then storing 2^k n-bit strings explicitly. In the following, we now prove that the J-tree data structure admits fast query and update operations, i.e. operations that allow us to read one entry of the data structure, or update the value of one entry. While the query operation will be computable efficiently by evaluating only the decompressor D $O(k)$ times, the update operation will require evaluating some compressor C, and might "fail" in a certain well-defined sense. However, when the update does not fail, there is a deterministic algorithm which can verify, using $O(k)$ evaluations of D, that the resulting J-tree is in fact the true updated version of the original. We begin with the query or "access" operation:

▶ **Lemma 23** (J-tree Access Lemma). *Let $D^k[y]$ be a J-tree. Then for any $i \in \{0,1\}^k$ we can compute $D^k[y](i)$ in time $O(nk)$ given i, y and using $O(k)$ evaluations of D.*

Proof. This follows directly from the definition of $D^k[y]$. Letting $b_1, \ldots, b_k$ be the individual bits of i, we have that:

$$D^k[y](i) = D_{b_k} \circ D_{b_{k-1}} \circ \cdots \circ D_{b_1}(y)$$

So we can compute $D^k[y](i)$ by evaluating D k times successively starting with the input y. ◀

▶ **Definition 24** (J-tree Modifications). *Let $D^k[y]$, $D^k[y']$ be J-trees of depth k and code length n. We say that the relation* Modified(y, y', i, s, D) *holds if:*

1. $D^k[y'](i) = s$
2. *For all $i' \in \{0,1\}^k$, $i' \neq i$, $D^k[y'](i') = D^k[y](i')$*

In other words, Modified(y, y', i, s, D) asserts that the J-tree $D^k[y']$ represents a local modification of the J-tree $D^k[y]$ which changes its i^{th} value to s and leaves all other values the same.

▶ **Lemma 25** (J-tree Update Lemma). *There exist algorithms* Find *and* Verify *satisfying the following.*

Verify *takes as input a decompressor D (specified as an oracle) of code length n and stretch 2, a pair of seeds $y, y' \in \{0,1\}^n$, an index $i \in \{0,1\}^k$, and a value $s \in \{0,1\}^n$, and either accepts or rejects.* Verify *runs in deterministic time $O(nk)$ using $O(k)$ evaluations of D, and satisfies the property that if* Verify(y, y', i, s, D) *accepts, then* Modified(y, y', i, s, D) *must hold.*

Find *takes as input a compression scheme C, D of code length n and stretch 2 (again specified as oracles), a seed $y \in \{0,1\}^n$, an index $i \in \{0,1\}^k$, and a value $s \in \{0,1\}^n$. It either "succeedes" and outputs a string $y' \in \{0,1\}^n$, or else outputs* FAIL$\langle e \rangle$*, where $e \in \{0,1\}^{2n}$.* Find *runs in $O(nk)$ time using $O(k)$ evaluations of C and D, and satisfies the property that for every input y, i, s, C, D, one of the following holds:*

1. Find(y, i, s, C, D) *outputs a string y' such that* Verify(y, y', i, s, D) *accepts.*
2. Find(y, i, s, C, D) *outputs* FAIL$\langle e \rangle$*, where e is incompressible with respect to C, D.*

Proof. We start by defining the procedure Find. Given inputs y, i, s, C, D, Find begins by computing a sequence of k values $z_1, \ldots, z_k \in \{0,1\}^n$ as follows. Let $b_1, \ldots, b_k$ denote the bits of i in order. For each $j \in [k]$, we set

$$z_j = D[y](b_1 b_2 \ldots b_{j-1} \neg b_j)$$

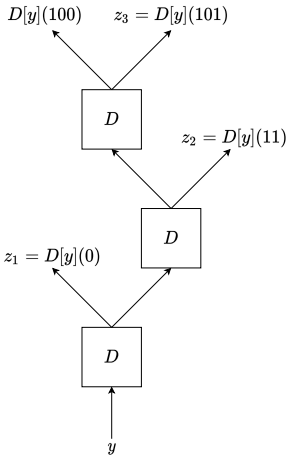

Figure 3 "Forward phase" of $\mathsf{Find}(y, 100, s, C, D)$ (where $k = 3$), during which the $z_1, \ldots, z_k$ are computed.

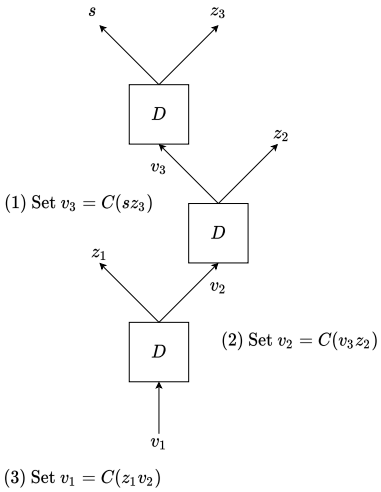

Figure 4 "Backward phase" of $\mathsf{Find}(y, 100, s, C, D)$ (where $k = 3$), during which the $v_1, \ldots, v_k$ are computed.

It is clear that the list of z_j be computed in $O(nk)$ time using $O(k)$ evaluations of D, by storing the intermediate values of $D[y](b_1 \cdots b_j)$ for each j and computing the z_j in increasing order of j.

Now, given this list of values, we compute a second sequence $v_1, \ldots, v_k \in \{0,1\}^n$. We compute the v_j for each $j \in [k]$ in decreasing order of k as follows. First we set:

$$v_k = \begin{cases} C(sz_k) & \text{if } b_k = 0 \\ C(z_k s) & \text{if } b_k = 1 \end{cases}$$

We then check that $D(C(sz_k)) = sz_k$ (resp. $D(C(z_k s)) = z_k s$). If this check fails we abort and return $\mathsf{FAIL}\langle sz_k \rangle$ (resp. $\mathsf{FAIL}\langle z_k s \rangle$). Now, for each $j \in \{k-1, k-2, \ldots, 1\}$, we set:

$$v_j = \begin{cases} C(v_{j+1} z_j) & \text{if } b_j = 0 \\ C(z_j v_{j+1}) & \text{if } b_j = 1 \end{cases}$$

Again, each time we evaluate C on some input, we check that that input is indeed compressible with respect to C, D, and if not then we return $\mathsf{FAIL}\langle e \rangle$ where e is the incompressible string we found. If we get to the end of this process without returning failure, we return the string v_1. This completes the description of the Find procedure, which overall requires at most $O(nk)$ time to compute using at most $O(k)$ evaluations of C and D. Figures 3 and 4 illustrate this procedure for a J-tree of depth 3.

We now describe the Verify procedure on input y, y', i, s, D, which simply verifies that a given string y' is a possible successful output of $\mathsf{Find}(y, i, s, C, D)$ for some C. Given y, y', i, s, D, again let $b_1, \ldots, b_k$ denote the bits of i in order. First, Verify iterates over each value of $j \in [k]$, and checks that $D[y'](b_1, \ldots, b_{j-1}, \neg b_j) = D[y](b_1, \ldots, b_{j-1}, \neg b_j)$. Next, it checks that $D[y'](b_1, \ldots, b_k) = D^k[y'](i) = s$. If all these conditions hold, the Verify procedure accepts, and otherwise it rejects. It is clear that these conditions can be verified in $O(nk)$ time using $O(k)$ evaluations of D, by storing the intermediate values $D[y](b_1, \ldots, b_j)$,$D[y'](b_1, \ldots, b_j)$ at each step.

We now show that if $\mathsf{Find}(y, i, s, C, D)$ succeeds and returns a string y', then $\mathsf{Verify}(y, y', i, s, D)$ accepts. By definition, if $\mathsf{Find}(y, i, s, C, D)$ doesn't fail, then it is able to compute some list of values $v_1, \ldots, v_k \in \{0,1\}^n$ such that for all $j < k$, $D_{b_j}(v_j) = v_{j+1}$ and $D_{\neg b_j}(v_j) = z_j$, and it returns v_1 as its output. So then we have that $D[v_1](b_1, \ldots, b_{j-1}, \neg b_j) = D[v_j](\neg b_j) = z_j$ for all $j \in [k]$. Further, we have that $D[v_1](b_1, \ldots, b_{k-1}, b_k) = D[v_k](b_k) = s$. So overall $\mathsf{Verify}(y, v_1, i, s, D)$ must accept if $\mathsf{Find}(y, i, s, C, D)$ succeeds and returns v_1.

It remains only to show that if $\mathsf{Verify}(y, y', i, s, D)$ accepts, then $\mathsf{Modified}(y, y', i, s, D)$ must hold. Recall that $\mathsf{Modified}(y, y', i, s, D)$ asserts that the two J-trees $D^k[y]$, $D^k[y']$ agree an all indices $i' \neq i$, and that $D^k[y'](i) = s$. If Verify accepts then this second condition holds trivially, since Verify explicitly checks that $D^k[y'](i) = s$ and rejects if this does not hold. Now consider the first condition of $\mathsf{Modified}$. Let $i' \in \{0,1\}^k$, $i' \neq i$. Let $\ell_1, \ldots, \ell_k$ denote the bits of i' in order, and let $t \in [k]$ be the smallest index such that $\ell_t \neq b_t$. By our assumption that Verify accepted, we have that

$$D[y](\ell_1, \ldots, \ell_t) = D[y](b_1, \ldots, b_{t-1}, \neg b_t) = D[y'](b_1, \ldots, b_{t-1}, \neg b_t) = D[y'](\ell_1, \ldots, \ell_t)$$

But by definition we also know that

$$D[y](\ell_1, \ldots, \ell_t, \ell_{t+1}, \ldots, \ell_k) = D[D[y](\ell_1, \ldots, \ell_t)](\ell_{t+1}, \ldots, \ell_k)$$

and similarly

$$D[y'](\ell_1, \ldots, \ell_t, \ell_{t+1}, \ldots, \ell_k) = D[D[y'](\ell_1, \ldots, \ell_t)](\ell_{t+1}, \ldots, \ell_k)$$

so if $D[y](\ell_1, \ldots, \ell_t) = D[y'](\ell_1, \ldots, \ell_t)$ then it must be that $D[y](\ell_1, \ldots, \ell_k) = D[y'](\ell_1, \ldots, \ell_k)$. So we have established that $D^k[y](i') = D^k[y'](i')$, which completes the proof. ◀

In addition to the update and access lemmas, we will need the following "initialization" lemma, which lets us efficiently set all leaves of a J-tree to a common value in time proportional to its depth:

▶ **Lemma 26** (J-Tree Initialization Lemma)**.** *There is an algorithm* Initialize *which takes as input a compression scheme C, D of code length n and stretch 2 (specified as oracles), a value $s \in \{0,1\}^n$, and a depth parameter k. It either succeeds and returns a seed $y \in \{0,1\}^n$ such that for all $i \in \{0,1\}^k$, $D^k[y](i) = s$, or else outputs $\mathsf{Fail}\langle e \rangle$ where e is incompressible with respect to C, D. $\mathsf{Initialize}(s, k, C, D)$ runs in time $O(nk)$ using $O(k)$ evaluations of C an D.*

Finally, we utilize the following "iterated compression" lemma.

▶ **Lemma 27** (J-Tree Iterated Compression Lemma)**.** *Let C, D be a compression scheme of code length n, and let $S = (s_1, s_2, \ldots, s_\ell)$ be a sequence of strings, where $s_i \in \{0,1\}^n$. There exists an algorithm* Iter-Compress *running in time $\mathrm{poly}(n, \ell)$ and using $\mathrm{poly}(\ell)$ evaluations of C and D which, given C, D and S, either "succedes" and outputs a seed $y \in \{0,1\}^n$ such that $D^{\lceil \log \ell \rceil}[y](i) = s_i$ for all $i \in \{0,1\}^{\lceil \log \ell \rceil}, i \leq \ell$, or else "fails" and outputs a string e which is incompressible with respect to C, D.*

The proofs of both the iterated compression and initialization lemmas follow the same format as the update lemma, and can be found in the full version.

5 Low-Space Simulations

In this section we prove our main set of theorems. In each case, we show how to simulate an exponential time computation with a drastic reduction in certain resources, using short oracle calls to some uniform compression scheme. We then show that either this simulation is successful, or there is an explicit construction algorithm that prints incompressible strings for this compression scheme.

5.1 Proofs of Main Theorems

▶ **Theorem 28.** *Let $\mathcal{C}, \mathcal{D} = \{C_n\}_{n\in\mathbb{N}}, \{D_n\}_{n\in\mathbb{N}}$ be a uniform compression scheme. Then one of the following must hold:*

1. *There is polynomial time algorithm with oracle access to $\mathcal{C}, \mathcal{D}$ which, for infinitely many n, outputs an incompressible string for C_n, D_n on input 1^n.*
2. *For every exponential time bound T, every language $L \in$ **RAM-TIME**$[T(n)]$, and every $\epsilon > 0$, there is 1-tape Turing machine with oracle access to $\mathcal{C}, \mathcal{D}$ which decides L in time $T(n)^{1+\epsilon}$, uses space at most $T(n)^\epsilon$, and makes oracle calls of length at most $T(n)^\epsilon$.*

Proof. Let $L \in$ **RAM-TIME**$[T(n)]$, and let $\mathcal{M}$ be the deterministic random access oracle machine witnessing this inclusion. Let $\mathcal{C}, \mathcal{D}$ be a uniform compression scheme. By Lemma 15, we can assume C_n, D_n have stretch 2 for all n. We will define a "simulator machine" which attempts to decide L using low space and small oracle calls to $\mathcal{C}, \mathcal{D}$. We then define a second "checker machine" which checks the work of the simulator. We show that whenever the simulator fails to decide L, the checker will be able to witness this failure in the form of an incompressible string. We will thus conclude that if simulation of L fails for infinitely many inputs, the "checker" will constitute an explicit construction algorithm which prints incompressible strings for $\mathcal{C}, \mathcal{D}$.

Step 1 – Defining the Simulator. Let $0 < \epsilon < 1$ be a fixed rational constant. We define a machine $\mathcal{S}_\epsilon$ which will attempt to efficiently simulate $\mathcal{M}$ using low space, short $\mathcal{C}, \mathcal{D}$-oracle queries, and which operates on a 1-tape oracle Turing machine. Let $x \in \{0,1\}^n$ be an input. For the remainder of thi s section we will keep a particular input length n fixed in our mind and thus drop the dependence of other terms on n in our notation; in particular we will use the abbreviation $T = T(n)$. Our machine will now fix a particular instance of $\mathcal{C}, \mathcal{D}$, in particular $C_{\lceil 2^{\epsilon n}\rceil}, D_{\lceil 2^{\epsilon n}\rceil}$; again we simplify our notation from here on and simply write C, D for this scheme. By definition of a uniform compression scheme, we see that the code length of C, D will by $poly(2^{\epsilon n}) = T(n)^{O(\epsilon)}$, and by assumption its message length is twice its code length. We will use W to denote its code length for the remainder of the proof.

We start by invoking Lemma 5, which lets us assume without loss of generality that $\mathcal{M}$ uses its first T cells of RAM and uses at most $O(\log T)$ space on its linear tapes. This simplification will come at a $\log T$ multiplicative cost to run time, which is negligible here. Now, our simulating machine will initialize a variable $\texttt{Mem} \in \{0,1\}^W$ which will be used as a seed in a J-tree with decompressor D and depth $\lceil \log T \rceil$. From now on we will fix $k = \lceil \log T \rceil$. The simulation $\mathcal{S}_\epsilon$ will run in T phases, and will maintain the invariant that if $\mathcal{M}$'s i^{th} memory cell has value $b \in \{0,1\}$ at the start of $\mathcal{M}$'s t^{th} time step, then $D^k[\texttt{Mem}](i) = b^W$ at the start of $\mathcal{S}_\epsilon$'s t^{th} phase. Aside from $\texttt{Mem}$, $\mathcal{S}_\epsilon$ will also explicitly store a copy of $\mathcal{M}$'s linear tapes (which have total length $O(k)$), the input x, and descriptions of $\mathcal{M}$'s state and tape-head pointers, each requiring at most k bits. Thus the total space required to store all local variables between phases in the simulation is at most $O(W(n)^2 + n) = T(n)^{O(\epsilon)}$ (recall that T is an exponential time bound).

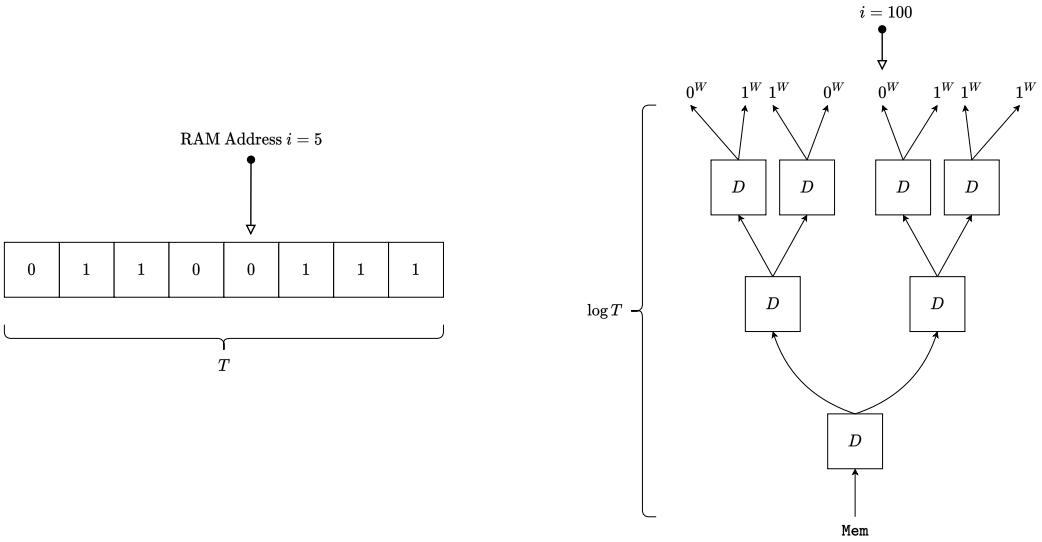

Figure 5 Simulating a RAM memory with a J-tree. While the RAM memory on the left requires T bits to store explicitly, the virtual RAM on the right can be stored with $|\texttt{Mem}| = W = T^{O(\epsilon)}$ bits.

At the start of the first phase, $\mathcal{S}_\epsilon$ initializes Mem to a value such that $D^k[\texttt{Mem}](i) = 0^W$ for all $i \in \{0,1\}^k$, which matches the initial state of $\mathcal{M}$'s RAM memory at the beginning of its computation. By Lemma 26, this can be accomplished in time $O(kW) = T^{O(\epsilon)}$ with oracle access to C, D; if the initialization procedure fails and returns an incompressible string for C, D, the simulation halts and rejects its input x. Now, in phase $t \in [T]$, $\mathcal{S}_\epsilon$ will simulate the t^{th} step of $\mathcal{M}$ as follows:

1. Read the values at the current tapehead positions on all linear tapes, and the current state of $\mathcal{M}$.
2. Based on these values, determine which of $\mathcal{M}$'s transition rules to apply. Every rule involves a state update, which we can perform manually as we explicitly store $\mathcal{M}$'s state. If the new state is an accept/reject state for $\mathcal{M}$, then $\mathcal{S}_\epsilon$ halts and accepts/rejects accordingly. Otherwise:
   a. First, say the rule only involves updates to the linear tapes. In this case we just carry out the rule explicitly, which requires at most $poly(k) = T^{o(\epsilon)}$ operations since the linear tapes have length $O(k)$.
   b. Next, say the rule involves a RAM operation. In this case we start by reading the entire contents of the addressing tape, which we denote $i \in \{0,1\}^k$. Now, if the operation is a Load, we compute the first bit of $D^k[\texttt{Mem}](i)$ and update the auxiliary tape at its current tapehead position to this value. If the operation is an Update, we read the value at the current position of the auxiliary tape, call it $s \in \{0,1\}$. We then compute $\mathsf{Find}(\texttt{Mem}, i, s^W, C, D)$. If this procedure fails and returns an incompressible string, $\mathcal{S}_\epsilon$ aborts its entire simulation and rejects its input. Otherwise, we update Mem to the value returned by this Find call.

If we get through all T phases without halting, we reject the input.

We now show that $\mathcal{S}_\epsilon$ operates within the required resource bounds. First, we note that all oracle calls are of length $W = T^{O(\epsilon)}$. Second, it is clear that the Find/Initialize operations and the evaluations of $D^k[\texttt{Mem}]$ can be carried out in $poly(W)$ space, since in particular they require at most $poly(W)$ time by Lemmas 23, 25, and 26. So the space used within a phase is at most $poly(W) = T^{O(\epsilon)}$ and the size of oracle calls are bounded identically. To bound

the time complexity, we note that by the same arguments each phase can be completed in time $T^{O(\epsilon)}$, and overall there are T phases, so the total time complexity is $T^{1+O(\epsilon)}$. In the above description of each phase, we were informal about the number of work tapes required to carry out these computations. However, since any multi-tape machine running in space S and time T can be simulated on a one-tape machine in time $poly(S, T)$ and space $poly(S)$, we see that we can modify the algorithm within each phase to operate on a 1-tape machine with at most a polynomial blowup in space and time. So the above bounds still hold on a 1-tape machine, where the space is bounded by $poly(T^\epsilon) = T^{O(\epsilon)}$, and the time within each phase is bounded identically.

Step 2 – If Find Never Fails Then the Simulator Works. We now show that if $\mathcal{S}_\epsilon$ completes its computation on an input x without any Initialize or Find operation failing, then $\mathcal{S}_\epsilon$ accepts x if and only if $x \in L$. To prove this, we show by induction on $t \in [T]$ that at the beginning of the t^{th} phase of $\mathcal{S}_\epsilon$'s simulation on x, $\mathcal{S}_\epsilon$'s simulated configuration matches the configuration of $\mathcal{M}$ on input x at the beginning of time step t. This is clearly true for $t = 1$, since at the start of the simulation we initialize explicit representations of $\mathcal{M}$'s linear tapes to all zeroes, and do the same for the RAM memory using the Initialize operation, which succeeds by assumption.

Now assume the inductive hypothesis up to step t. So at the beginning of this time step on input x, $\mathcal{M}$'s linear tapes, tapehead pointers, machine state and RAM memory at the start of time step t are faithfully represented by $\mathcal{S}_\epsilon$ at the start of phase t. Thus, $\mathcal{S}_\epsilon$ is able to choose the correct transition rule to apply during this phase. It then updates the linear tapes, tapehead pointers, and machine state accordingly; all of these are stored explicitly so $\mathcal{S}_\epsilon$ has no potential for failure here. Next, if $\mathcal{M}$ uses a Load operation at this step, since we assumed the previous state of the virtual RAM is accurate, the correct value will be found by $\mathcal{S}_\epsilon$. Finally, if $\mathcal{M}$ uses a Update operation at this step, $\mathcal{S}_\epsilon$ makes the associated call to the Find procedure. By Lemma 25, if this does not fail, then the state of the virtual RAM after this step will accurately represent $\mathcal{M}$'s ram at the end of time step t. This completes the inductive case.

Step 3 – Failure of the Simulator Witnesses the Pigeonhole Principle for $\mathcal{C}, \mathcal{D}$. We now show that if the above simulation $\mathcal{S}_\epsilon$ fails to decide L for infinitely many inputs, then there is an algorithm which, given 1^n, outputs an incompressible string for C_n, D_n in polynomial time using oracles for $\mathcal{C}, \mathcal{D}$, for infinitely many n.

In particular, assume that there is an infinite set $R \subseteq \mathbb{N}$ such that for all $n \in R$, $\mathcal{S}$ makes a mistake on some length n input in its attempt to decide L. Recall that $\mathcal{S}_\epsilon$ uses the compression scheme $C_{\lceil 2^{\epsilon n} \rceil}, D_{\lceil 2^{\epsilon n} \rceil}$ in its simulation on inputs of length n. Now, let $\mathbb{I} = \{\lceil 2^{\epsilon n} \rceil \mid n \in R\}$. We will give a construction algorithm that succeeds for all input lengths in $\mathbb{I}$.

Given an input 1^m, our construction algorithm $\mathcal{H}$ operates as follows. It starts by computing an n such that $\lceil 2^{\epsilon n} \rceil = m$; by construction we see $n = O(\log m)$. Next, $\mathcal{H}$ runs the simulation $\mathcal{S}_\epsilon$ on all inputs of length n one after the other; by construction we see that $\mathcal{S}_\epsilon$ uses the same compression scheme C_m, D_m on all such inputs. During each simulation, if $\mathcal{S}_\epsilon$ fails on some Find or Initialize operation and returns an incompressible string for C_m, D_m, $\mathcal{H}$ halts and outputs that string. If $\mathcal{H}$ gets through all inputs of length n without any operation failing, it outputs something arbitrary, say 1^m. By definition we see that when $m \in \mathbb{I}$, $\mathcal{H}$ cannot get through all simulations without $\mathcal{S}_\epsilon$ failing, since by definition of $\mathbb{I}$ we know that $\mathcal{S}_\epsilon$ fails to compute L on some input of length n, and by the previous section we know this can

only happen when $\mathcal{S}_\epsilon$ fails on some Find/Initialize operation. So overall we have that $\mathcal{H}$ runs in $poly(2^n) = 2^{O(n)} = 2^{O(\log m)} = poly(m)$ time, and successfully finds an incompressible string for C_m, D_m on input 1^m for infinitely many m.

Step 4 – Wrapping Things Up. We now have that for any language $L \in$ **RAM-TIME**$[T(n)]$ and every $\epsilon > 0$, there is a machine S_ϵ running in time $T(n)^{1+O(\epsilon)}$, space $T^{O(\epsilon)}$, and making $\mathcal{C}, \mathcal{D}$ oracle calls of length $T^{O(\epsilon)}$, such that one of the following holds:

1. $\mathcal{S}_\epsilon$ correctly decides L on all but finitely many inputs.
2. There is a polynomial time construction algorithm which finds incompressible strings for C_n, D_n given oracles for $\mathcal{C}, \mathcal{D}$, which works for infinitely many inputs.

Clearly if the first possibility holds then we can get a simulation of the same complexity which decides L exactly. In addition, if the first possibility holds for all $\epsilon > 0$, then we can replace the $O(\epsilon)$ terms with ϵ as we take ϵ to zero. This yields the stated theorem. ◀

We now prove a variant of the above theorem, which allows us to replace the oracle for the compressor $\mathcal{C}$ in our low-space simulation with nondeterminism. This will be relevant for schemes $\mathcal{C}, \mathcal{D}$ where $\mathcal{C}$ is significantly harder to compute then $\mathcal{D}$.

▶ **Theorem 29.** *Let T be an exponential time bound, and let $\mathcal{C}, \mathcal{D}$ be a uniform compression scheme.*

Then one of the following must hold:

1. *There is polynomial time algorithm with oracle access to $\mathcal{C}, \mathcal{D}$ which, for infinitely many n, outputs an incompressible string for C_n, D_n on input 1^n.*
2. *For every language $L \in$ **RAM-TIME**$[T(n)]$ and every $\epsilon > 0$, there is nondeterministic 1-tape Turing machine with oracle access to $\mathcal{D}$ which decides L in time $T(n)^{1+\epsilon}$, uses space at most $T(n)^\epsilon$, and makes at most $T(n)^\epsilon$ nondeterministic guesses on all computation paths.*

Proof. We follow the same simulation $\mathcal{S}_\epsilon$ of some language L in **RAM-TIME**$[T(n)]$, with a slight modification. The key observation is the following: the procedure Find is proven to work with access to both $\mathcal{C}$ and $\mathcal{D}$ oracles by Lemma 25. However, recall that Lemma 25 also defines separate procedure Verify, which is able to verify that a certain J-tree seed is the updated form of another, and this verification only needs the $\mathcal{D}$ oracle. Recall also the key property relating Find and Verify, which says that Verify will accept any successful output of a Find operation.

Thus, if we make our simulator $\mathcal{S}_\epsilon$ nondeterministic, it can replace the Find operation by a nondeterministic guess as to the new value of the seed Mem during each phase, and then use Verify to check that this guess is correct, which requires only the $\mathcal{D}$ oracle. This immediately gives us a low space nondeterministic simulator $\mathcal{S}_\epsilon$ with similar properties as that in the proof of Theorem 28. However, the total nondeterminism used by this simulation is $T(n)^{1+O(\epsilon)}$, since it has to guess a seed of length $T^{O(\epsilon)}$ during each of the T phases.

To get away with $T^{O(\epsilon)}$ bits of nondeterminism, we use the following trick[4]. At the beginning of $\mathcal{S}_\epsilon$'s simulation, it guesses a single seed for a separate J-tree of the same code length, call this seed Up. Now, during phase t, to simulate an Update operation which sets RAM cell $i \in \{0,1\}^k$ to value $s \in \{0,1\}$, we first set $\texttt{Mem}' = D^k[\texttt{Up}](t)$, and then check if $\mathsf{Verify}(\texttt{Mem}, \texttt{Mem}', i, s^W, D)$ holds. If so we then set $\texttt{Mem} = \texttt{Mem}'$ and continue to the next phase,

[4] This trick is essentially the "easy witness method" of [11].

and if not then we halt the simulation and reject outright. It is straightforward to see that this simulation runs in time $T(n)^{1+O(\epsilon)}$, uses space $T(n)^{O(\epsilon)}$, and guesses at most $T(n)^{O(\epsilon)}$ nondeterministic bits.

By the same arguments as in Theorem 28, we have that if every Verify call accepts during $\mathcal{S}_\epsilon$'s computation on input x, then $\mathcal{S}_\epsilon$ correctly decides if $x \in L$. To finish the proof, it suffices to show that given some x such that $\mathcal{S}_\epsilon$ fails to decide if $x \in L$, we can construct an incompressible string for $C_{2^{\epsilon n}}, D_{2^{\epsilon n}}$ in $poly(T(n))$ time with $\mathcal{C}, \mathcal{D}$ oracles; from here the theorem follows directly using the arguments at the end of the proof of Theorem 28.

So, say we have such an x. Since our simulation errs on the side of rejecting, we know that $x \in L$ but $\mathcal{S}_\epsilon$ rejects x. We start by running the *deterministic* low space simulation from Theorem 28 (corresponding to L, ϵ) on x, which can be done in $poly(T(n))$ time with $\mathcal{C}, \mathcal{D}$ oracles. During this simulation, we keep track of all updated memory seeds constructed using Initialize and Find. If at any step a Find or Initialize procedure fails, we find an incompressible string and output it. Otherwise, we have found a sequence of seeds $\texttt{Mem}_1, \ldots, \texttt{Mem}_T \in \{0,1\}^W$ (where W is the code length used by the simulation), such that $\texttt{Mem}_{t+1}$ represents a correct update to the RAM memory previously represented by $\texttt{Mem}_t$ after time step t. Now, we use the Iter-Compress procedure to construct a seed $\texttt{Up} \in \{0,1\}^W$ such that for all $t \in \{0,1\}^k$, $D^k[\texttt{Up}](t) = \texttt{Mem}_t$. By Lemma 27, in $poly(T(n))$ time we can either find such a seed Up, or else find an incompressible string for our scheme. But if such a string Up exists, this is precisely the nondeterministic guess which would cause our nondeterministic simulation $\mathcal{S}_\epsilon$ to accept x. So by assumption that it rejects x, if we get to this point then Iter-Compress must find an incompressible string. ◀

Finally, we show that if the explicit construction algorithm in the above theorem is also granted access to an **NP** oracle, we can get the same result, but where we simulate languages in the larger class $\mathbf{NTIME}[T(n)]$.

▶ **Theorem 30.** *Let T be an exponential time bound, and let $\mathcal{C}, \mathcal{D}$ be a uniform compression scheme.*

Then one of the following must hold:

1. *There is polynomial time algorithm with oracle access to $\mathcal{C}, \mathcal{D}$, and SAT which, for infinitely many n, outputs an incompressible string for C_n, D_n on input 1^n.*
2. *For every language $L \in \mathbf{NTIME}[T(n)]$ and every $\epsilon > 0$, there is nondeterministic 1-tape Turing machine with oracle access to $\mathcal{D}$ which decides L in time $T(n)^{1+\epsilon}$, uses space at most $T(n)^\epsilon$, and makes at most $T(n)^\epsilon$ nondeterministic guesses on all computation paths.*

Proof. This will follow quite directly from the proof of the previous theorem. Let $\mathcal{M}$ be some $\mathbf{NTIME}[T(n)]$ machine which we are attempting to simulate. At the outset of its computation, in addition to guessing one seed Up which encodes a sequence of RAM-seed updates, our new simulator $\mathcal{S}_\epsilon$ also guesses a second seed Wit which will encode the nondeterministic choices made by $\mathcal{M}$ during its computation. In particular, after guessing this seed, at each phase t of our simulation the simulator reads the first $O(1)$ bits of $D^k[\texttt{Wit}](t)$, and uses this to determine which transition rule to apply at that step (in a nondeterministic machine there can be $O(1)$ such rules, which will be smaller then the code length of the decompressor for sufficiently large input lengths). Otherwise we run the simulation exactly as in the above proof.

By the same reasoning as in the previous proof, we see that if this machine accepts an input x then $x \in L$ (since the simulation errs on the side of rejection), but it is possible that $x \in L$ and the simulation fails to determine this. In such a case, the only possibility is that for all sequences $z \in \{0,1\}^{O(T)}$ of nondeterministic guesses causing $\mathcal{M}$ to accept x, there is no pair $\texttt{Up}, \texttt{Wit} \in \{0,1\}^W$ such that Wit encodes z (as described above), and Up encodes a

sequence of valid RAM seeds representing the deterministic computation of $\mathcal{M}$ on x with witness z. Thus, if we have some input x on which our simulation fails, we can use an **NP** oracle to compute a witness z causing $\mathcal{M}$ to accept x. We then run the deterministic low space simulation $\mathcal{M}$ on x with witness z, and either find an incompressible string for $\mathcal{C}, \mathcal{D}$, or else find a sequence of valid RAM seeds $\texttt{Mem}_1, \ldots, \texttt{Mem}_T$ for this computation. We then proceed as in the previous Theorem, attempting to compress the $\texttt{Mem}_t$ to a single seed `Up` and the bits of z to another seed `Wit`, both using the Iter-Compress procedure. By assumption that our main nondeterministic simulation rejected x, if we get to this point we know that one of these Iter-Compress procedures must return an incompressible string. ◀

5.2 Implications

Consider the following three hypotheses:

▶ **Hypothesis 1.** *There exists some exponential time bound T and some $\epsilon > 0$ such that:*

$$\mathbf{RAM\text{-}TIME}[T(n)] \not\subseteq \mathbf{1\text{-}TISP}[T(n)^{1+\epsilon}, T(n)^{\epsilon}]$$

▶ **Hypothesis 2.** *There exists some exponential time bound T and some $\epsilon > 0$ such that:*

$$\mathbf{NTIME}[T(n)] \not\subseteq \mathbf{1\text{-}NTISPG}[T(n)^{1+\epsilon}, T(n)^{\epsilon}, T(n)^{\epsilon}]$$

▶ **Hypothesis 3.** *There exists some exponential time bound T and some $\epsilon > 0$ such that:*

$$\mathbf{RAM\text{-}TIME}[T(n)] \not\subseteq \mathbf{1\text{-}NTISPG}[T(n)^{1+\epsilon}, T(n)^{\epsilon}, T(n)^{\epsilon}]$$

The first Hypothesis asserts that deterministic exponential time RAM computations cannot be simulated on 1-tape machines using low space and near-linear blowup in time. The second Hypothesis asserts roughly the same for nondeterministic computations, claiming that such a simulation cannot occur which simultaneously uses a small amount of nondeterminism. Recall that for nondeterministic time, the multi-tape and RAM models are roughly equivalent, which is why we don't make a distinction here on the left-hand side. Finally, the third hypothesis says that deterministic RAM computations cannot be recognized by machines with short "proofs" verifiable in low space and near-linear time on a 1-tape machine.

While 1 and 2 seem incomparable and both quite reasonable, we see that 3 is formally stronger then the previous two, and we have significantly less intuition as to whether or not it should be true. However, recall from Section 1.3 that when the time bound T is linear, all three of these hypotheses are known to hold unconditionally (indeed they hold for *all* $\epsilon < 1$). In any case, the results of the previous section give us the following:

▶ **Theorem 31.** *If Hypothesis 1 holds, then for any uniform compression scheme $\mathcal{C}, \mathcal{D}$ where both $\mathcal{C}$ and $\mathcal{D}$ are computable in polynomial time, there is a polynomial time algorithm which, given 1^n, prints an incompressible string for C_n, D_n for infinitely many n.*

▶ **Theorem 32.** *If Hypothesis 2 holds, then for any uniform decompressor $\mathcal{D}$ computable in polynomial time, there exists there is a polynomial time $\mathbf{NP}$-oracle algorithm which, given 1^n, prints an empty pigeonhole for D_n for infinitely many n. In particular, $\mathbf{E}^{\mathbf{NP}} \not\subseteq \mathbf{size}[2^n/2n]$.*

▶ **Theorem 33.** *If Hypothesis 3 holds, then for any uniform compression scheme $\mathcal{C}, \mathcal{D}$ where $\mathcal{D}$ is computable in polynomial time, there is a polynomial time algorithm using a $\mathcal{C}$ oracle which, given 1^n, prints an incompressible string for C_n, D_n for infinitely many n.*

In each case, the stated result follows from the fact that oracle calls of length $T^{O(\epsilon)}$ to a problem computable in polynomial time can be simulated directly without effecting the resource bounds of the simulation as we take $\epsilon \to 0$. To illustrate the use of these theorems, we first consider their implications for the compression schemes related to non-uniform complexity measures described in Section 3.2.1. In these cases, we have some complexity measure on strings/truth tables, such as circuit complexity, formula size, or K^{poly} complexity, and wish to construct strings/truth tables of high complexity (for infinitely many input lengths), which we call the "construction problem." In each case there is an associated "compression problem" in **FNP**, where we are given a string/truth table and wish to find a small circuit/formula/program computing it if one exists. For these sorts of problems, we now have the following:

▶ **Corollary 34.** *For each of the above mentioned uniform compression schemes related to non-uniform complexity measures, we have:*

1. *If Hypothesis 1 holds, then an efficient algorithm for the compression problem implies an efficient algorithm for the construction problem.*
2. *If Hypothesis 2 holds, then there is an efficient* **NP***-oracle algorithm for the construction problem.*
3. *If Hypothesis 3 holds, then there is an efficient algorithm for the construction problem using an arbitrary oracle for the compression problem. In other words, the construction problem reduces to the compression problem.*

We see here that the conclusion of the third implication implies the conclusion of the previous two. As noted in the introduction, the second implication can be proved by simpler techniques, utilizing the "Easy Witness Lemma." We give an alternate proof using this method in the full version for the interested reader. However, for the first and third implication, the J-tree update lemma seems necessary.

The case of large prime construction does not quite fit in with the others, as both the compressor and decompressor given in Theorem 21 require a factoring oracle. In addition, the second implication of Corollary 34 above is trivial when applied to the construction of large primes, since primality testing is in **P** [1]. However we can still derive the following from Theorem 31:

▶ **Corollary 35.** *If Hypothesis 1 holds, then a polynomial time algorithm for factoring implies a polynomial time algorithm to construct* $16n$*-bit primes of magnitude* $> 2^n$ *(for infinitely many* n*).*

More generally we can conclude the following directly from Theorem 28:

▶ **Corollary 36.** *One of the following is true:*

1. *For every exponential time bound* T *and every* $\epsilon > 0$*, every language decidable in time* $T(n)$ *on a RAM machine can be decided in time* $T(n)^{1+\epsilon}$ *and space* $T(n)^{\epsilon}$ *by a 1-tape machine with a factoring oracle, which makes oracle calls of length at most* $T(n)^{\epsilon}$*.*
2. *There is a polynomial time algorithm with a factoring oracle that generates* $32n$*-bit primes of magnitude* $> 2^n$ *for infinitely many* n*.*

6 BPP, TFNP, and the Weak Pigeonhole Principle

Throughout this paper we have studied the problem of finding incompressible strings for *uniform* compression schemes. As mentioned in Section 3, it would also be natural to study the more general search problem, where a compression scheme of a fixed message/code length is given as input in the form of a boolean circuit:

▶ **Definition 37.** Lossy Code *is the following problem: given circuits* $C\colon \{0,1\}^n \to \{0,1\}^{n-1}$ *and* $D\colon \{0,1\}^{n-1} \to \{0,1\}^n$*, find some* x *such that* $D(C(x)) \neq x$*.*

It follows from the definition that Lossy Code lies in **TFNP** (more specifically the class **PPP** defined in [18]), and reduces to the problem Empty studied in [13] and [14]. Further, we have seen that Lossy Code admits a randomized algorithm that outputs a solution with high probability. Since solutions can be verified efficiently, this fits it into a family of search problems studied by Goldreich [4], whose results imply the following:

▶ **Lemma 38.** Lossy Code *is polynomial time Turing reducible to* CAPP*.*

Here, CAPP denotes the canonical complete problem for **prBPP**, where we are given as input a circuit $E\colon \{0,1\}^n \to \{0,1\}$, and must output an approximation of $Pr_x[E(x) = 1]$ that is accurate to within $\pm\frac{1}{6}$. A natural question is whether the converse holds, i.e. whether CAPP reduces deterministically to Lossy Code. As hinted towards in Section 3.2, a polynomial time algorithm for Circuit Synthesis would imply such a reduction, but an unconditional reduction would be a major breakthrough as it would place $\mathbf{BPP} \subseteq \mathbf{NP}$. We show here that if the compressor C in Lossy Code is allowed to be randomized, the associated total search problem is in fact Turing-equivalent to CAPP, and thus complete for **prBPP**. The problem is defined formally as follows:

▶ **Definition 39.** R-Lossy Code *is the following problem: given* $C\colon \{0,1\}^n \times \{0,1\}^m \to \{0,1\}^{n-1}$*,* $D\colon \{0,1\}^{n-1} \to \{0,1\}^n$*, find some* $x \in \{0,1\}^n$ *such that* $Pr_r[D(C(x,r)) = x] < \frac{1}{2}$*.*

To clarify, this is precisely the same problem as Lossy Code, except that the "compressor circuit" C now uses some random coins. In this case, we seek a string which has a $< \frac{1}{2}$ probability of being recoverable from its description (where the probability is taken over the random coins of the compressor C). Just like Lossy Code this search problem is total, but verifying a solution is no longer in **P**. Indeed, it may be **PP**-hard to determine if any particular string is a solution, but there is a efficient randomized verification procedure that accepts only valid solutions, and accepts a random solution with high probability.

In the full version we show the following:

▶ **Theorem 40.** R-Lossy Code *and* CAPP *are Turing reducible to one another in deterministic polynomial time.*

An interesting takeaway from the proof of the above theorem is the following: choosing a good fixing of the "leftover bits" in Yao's lemma is in some sense a *universal* probabilistic search problem, since derandomizing this step would give a reduction from **prBPP** to Lossy Code $\in$ **TFNP** (and in particular would imply $\mathbf{BPP} \subseteq \mathbf{NP}$). More formally:

▶ **Corollary 41.** *Consider the following promise problem: we are given a circuit* $E\colon \{0,1\}^n \to \{0,1\}$*, a matrix* $A \in \{0,1\}^{m \times n}$*, and an index* $i \in [n]$*, such that* A *fails Yao's next-bit test with respect to* E *at index* i *with bias* $\epsilon = poly(\frac{1}{m})$*, and we must produce some* $z \in \{0,1\}^{n-i}$ *such that fixing the last* $n - i$ *bits of* E *to* z *preserves a non-negligible bias for this next-bit test. If this problem can be solved in deterministic polynomial time, then* **prBPP** *reduces to* Lossy Code*.*

References

1 Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Ann. of Math*, 2:781–793, 2002.

2 Lance Fortnow. Time–space tradeoffs for satisfiability. *Journal of Computer and System Sciences*, 60(2):337–353, 2000. doi:10.1006/jcss.1999.1671.

3 Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. ACM*, 52(6):835–865, November 2005. doi:10.1145/1101821.1101822.

4 Oded Goldreich. *In a World of P=BPP*, pages 191–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-22670-0_20.

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

6 Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at Botik '89*, pages 108–118, Berlin, Heidelberg, 1989. Springer-Verlag.

7 Russell Impagliazzo, Valentine Kabanets, and Avi Wigderson. In search of an easy witness: exponential time vs. probabilistic polynomial time. *Journal of Computer and System Sciences*, 65(4):672–694, 2002. Special Issue on Complexity 2001. doi:10.1016/S0022-0000(02)00024-7.

8 Russell Impagliazzo and Avi Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 220–229, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258533.258590.

9 Emil Jeřábek. Dual weak pigeonhole principle, boolean complexity, and derandomization. *Annals of Pure and Applied Logic*, 129(1):1–37, 2004. doi:10.1016/j.apal.2003.12.003.

10 Emil Jeřábek. On independence of variants of the weak pigeonhole principle. *Journal of Logic and Computation*, 17(3):587–604, 2007.

11 Valentine Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. *Journal of Computer and System Sciences*, 63(2):236–252, 2001. doi:10.1006/jcss.2001.1763.

12 Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 73–79, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/335305.335314.

13 Robert Kleinberg, Oliver Korten, Daniel Mitropolsky, and Christos Papadimitriou. Total Functions in the Polynomial Hierarchy. In James R. Lee, editor, *12th Innovations in Theoretical Computer Science Conference (ITCS 2021)*, volume 185 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITCS.2021.44.

14 Oliver Korten. The hardest explicit construction. In *62nd Annual Symposium on Foundations of Computer Science*, 2021.

15 Wolfgang Maass. Quadratic lower bounds for deterministic and nondeterministic one-tape turing machines. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 401–408, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800057.808706.

16 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

17 Igor C. Oliveira and Rahul Santhanam. Pseudodeterministic constructions in subexponential time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 665–677, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3055399.3055500.

18 Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994. doi:10.1016/S0022-0000(05)80063-7.

19 J. Paris, A. Wilkie, and Alan R. Woods. Provability of the pigeonhole principle and the existence of infinitely many primes. *J. Symb. Log.*, 53:1235–1244, 1988.

20 D. H. J. Polymath. Deterministic methods to find primes. *arXiv*, 2010. doi:10.48550/ARXIV.1009.3956.

21 J.M. Robson. Deterministic simulation of a single tape turing machine by a random access machine in sub-linear time. *Information and Computation*, 99(1):109–121, 1992. doi:10.1016/0890-5401(92)90026-C.

22 Ryan Williams. Inductive time-space lower bounds for sat and related problems. *Computational Complexity*, 15:433–470, 2005.

23 Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM Journal on Computing*, 42(3):1218–1244, 2013. doi:10.1137/10080703X.

24 Andrew C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 1982. doi:10.1109/SFCS.1982.45.