Electronic Colloquium on Computational Complexity, Report No. 25 (2022)

# Efficient Low-Space Simulations From the Failure of the Weak Pigeonhole Principle

Oliver Korten *

**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 space/time 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 space/time 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 a more general 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.

*Department of Computer Science, Columbia University. Email: oliver.korten@columbia.edu



ISSN 1433-8092

# 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 [Pap94], 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 [Yao82]. 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 $f : \{0,1\}^n \to \{0,1\}^{2n}$ 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 [Kor21] it is shown that a broad collection of such problems can be reduced to the following more general task: given some efficiently computable function $f : \{0,1\}^{n-1} \to \{0,1\}^n$, find an $n$-bit string outside the range of $f$. The existence of a solution is guaranteed by the "dual weak pigeonhole principle," and 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, which would imply that there are efficient constructions of truth tables of high circuit complexity. By [Kor21], this would in fact imply a generic "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 dual 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 $\mathcal{G} = \{g_n : \{0,1\}^n \to \{0,1\}^{n-1}\}_{n\in\mathbb{N}}$, $\mathcal{F} = \{f_n : \{0,1\}^{n-1} \to \{0,1\}^n\}_{n\in\mathbb{N}}$, 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 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

Let $\mathcal{C}, \mathcal{D} = \{C_n : \{0,1\}^{n-1} \to \{0,1\}^n, D_n : \{0,1\}^n \to \{0,1\}^{n-1}\}_{n\in\mathbb{N}}$. We call such a pair a "uniform generator/inverter pair," or "uniform g.i. pair" for short, where $\mathcal{C}$ is referred to as the generator and $\mathcal{D}$ as the inverter; the formal definition given in Section 3 is slightly more general, but we will use this simplification for now. We will say that a string $x \in \{0,1\}^n$ is "incompressible for $\mathcal{C}, \mathcal{D}$" if $C_n(D_n(x)) \neq x$. In Section 3.2, we show that a variety of 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 g.i. pair. In each case, the generator/inverter 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.

We then consider, for some $\mathcal{C}, \mathcal{D}$, whether there is an efficient explicit construction algorithm that produces incompressible strings for the pair. Our main results show that if there is no such algorithm, this implies the existence of a highly effective simulation of RAM machines using low space on a 1-tape machine, which makes short oracle queries to $\mathcal{C}, \mathcal{D}$. Depending on the precise features of the explicit construction algorithm presumed not to exist, our simulation will have varying properties. Ultimately these results are most interesting when stated in contrapositive form: they show that, assuming basic tradeoffs between time and space are necessary when converting a RAM algorithm to a 1-tape algorithm, a range of explicit construction problems must have efficient solutions.

Perhaps the cleanest of our results to state is the following:

**Theorem** (Theorem 5)**.** *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 g.i. pair $\mathcal{C}, \mathcal{D}$ where both the generator and inverter are computable in polynomial time, there is a polynomial time algorithm that produces incompressible strings for $C_n, D_n$ on input $1^n$ (for infinitely many $n$).*

To state our other 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:

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

Figure 1: Main Results

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), 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 g.i. pair we construct for this problem will require a factoring oracle for both the generator and inverter (while the above problems have a polynomial-time computable generator). In this case we get the following result:

**Theorem** (Corollary 2)**.** *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* $n$*.*

The problem of deterministically generating large primes has been investigated previously in several works, and was notably the subject of the Polymath 4 project. In the public discussion forums for this project, it was explicitly asked whether a polyomial time algorithm for factoring would help. More recently, Oliviera and Santhanam gave a subexponential time "pseudodeterministic" construction of large primes [OS17], using only the fact that primality is testable in **P** [AKS02] and that primes occur with non-negligible frequency.

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 generator/inverter pair $C, D$ 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 pair. 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 $C, D$.

### 1.2 Relation to Prior Work

**Karp-Lipton Theorems:** As mentioned above, the existence of explicit construction algorithms witnessing the weak pigeonhole principle is intimately connected to the circuit complexity of exponential time classes. Since our results prove the existence of such algorithms assuming certain space/time lower bounds for uniform classes, these results can be broadly placed within the framework of "Karp-Lipton type theorems:" theorems which derive non-uniform circuit lower bounds from separations between uniform classes [KL80]. Perhaps most relevant are the collapses due to Meyer (noted in [KL80]) and to Babai, Fortnow and Lund [BFL90], which show that for various exponential time classes such as $\mathbf{EXP}, \mathbf{NEXP}$, or $\mathbf{EXP}^{\mathbf{NP}}$, if these classes have small circuits then they in fact collapse to low levels of the polynomial hierarchy, and in particular must lie in **PSPACE**. When scaled upwards appropriately, these results would imply that, unless there are explicit constructions of truth tables with exponential circuit complexity, **E** can be decided in $2^{\epsilon n}$ space for any $\epsilon > 0$. Stated contrapositively, this means that if **E** *cannot* be solved in subexponential space, then circuit lower bounds (and thus a constructive proof of the pigeonhole principle) follow. At the surface this seems roughly similar to the results we obtained above. However, the simulation implied by these results would have time complexity roughly $2^{2^{\epsilon n}}$, which is far less efficient then what we show above. In this sense, the uniform separations that we assume are much weaker then what is required to apply known Karp-Lipton style theorems: we leave open the possibility that exponential time can be simulated in very low (even polynomial space), but simply posit that such a simulation would require a strongly superlinear blowup in time.

**The Easy Witness Lemma:** Perhaps the closest prior work to our results is the "Easy Witness Method," initiated by [Kab01] and furthered in [IKW02], 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 $\mathbf{E}^{\mathbf{NP}}$ has requires large circuits, we can

efficiently simulate $\mathbf{NTIME}[2^n]$ using limited nondeterminism by "guessing a small circuit." In Appendix B, we show that this old method (along with one other well-known tool) is in fact enough to prove the second implication in Figure 1, although this connection to space/time 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.

**Known Lower Bounds:** Finally, we cover the known results on time/space trade-offs and separations between the RAM and 1-tape models. A sequence of works ([For00], [FLvMV05], [Wil05] among others) have 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 on the RAM model). 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 hiearchy 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. When it comes to separating the RAM and 1-tape models, an old result of Maass [Maa84] shows that there are languages decidable in quasilinear time on a deterministic RAM machine which require $\Omega(n^2)$ time on a nondeterministic 1-tape machine; in particular, this separation applies to the problem of recognizing palindromes. If this separation could be scaled up to larger (exponential) time bounds, it would immediately imply all three hypothesis in Figure 1. However, Maass's proof is essentially a counting argument which crucially relies on the entropy of the input being comparable to total computation time, which is no longer the case for exponential time computations. Finally, a result of [Rob92] 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 $\log \log T(n)$.

### 1.3 History of the J-tree

As mentioned previously, the main tool used in our proofs is a data structure we call a "J-tree." The core construction underlying the J-tree has an intriguing history dating back to the late 1970's. In the course of a decade, this construction was used, seemingly independently, to prove three seminal results in cryptography and logic:

1. In 1979[1], Merkle showed how to iterate a cryptographically secure hash function $h : \{0,1\}^{2n} \to \{0,1\}^n$ in a tree-like fashion to construct a secure digital signature scheme whose signature size increases logarithmically in the number of messaged signed [Mer88].

2. In 1986, Goldwasser Goldreich and Micali showed how to iterate a pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^{2n}$ in the same tree-like manner into a family of functions $\{G_x : \{0,1\}^n \to \{0,1\}^n\}_{x \in \{0,1\}^n}$ whose input/output behaviours are indistinguishable from random, assuming $G$ is a secure PRG [GGM86].

3. In 1988, Paris Wilkie and Woods gave a novel proof of the weak pigeonhole principle in bounded arithmetic[PWW88]. Again, the same process of iterating a function $f : \{0,1\}^n \to \{0,1\}^{2n}$ is applied. Here, the function $f$ is assumed to be a bijection between $\{0,1\}^n$ and

[1] Although the cited work is dated to 1988, Merkle filed a patent for the concept about a decade earlier.

$\{0,1\}^{2n}$, and the iteration procedure is the first step towards establishing a contradiction in the relevant proof system.

Following [PWW88], Jeřábek used this same construction within bounded arithmetic to establish a deep connection between the weak pigeonhole principle and circuit complexity. This work most closely informs our perspective, hence the name "J-tree." In [Kor21], the techniques of Jeřábek were translated to the language of traditional complexity theory, which yield the following interpretation of this tree-like iteration process: given a circuit $C : \{0,1\}^n \to \{0,1\}^{2n}$, we can efficiently construct a new circuit $C^* : \{0,1\}^n \to \{0,1\}^{2^k n}$ satisfying the following: any string in the range of $C^*$ is a truth table of low circuit complexity, and any string outside the range of $C^*$ can be used to efficiently construct a string outside the range of $C$ using an **NP** oracle. If we have a generator/inverter pair $C : \{0,1\}^n \to \{0,1\}^{2n}, D : \{0,1\}^{2n} \to \{0,1\}^n$, the same argument lets us iterate these to a pair $C^*, D^*$ of input length $n$ and output length $2^k n$, where again $C^*$ always outputs low complexity truth tables, and now we have that any incompressible string for $C^*, D^*$ can be transformed in polynomial time (without an oracle) into an incompressible string for $C, D$.

Thus, if it is hard to witness the weak pigeonhole principle for $C, D$, then we will be able to take any string of length $2^k n$ and find a small circuit computing it by feeding it through $D^*$. A circuit can naturally be interpreted as a data structure storing an array subject to fast query operations: the truth table represents the contents of the array, and the circuit allows us to efficiently compute a particular bit in the truth table given its index. However, to use the J-tree to simulate RAM computations, we will also require fast update operations, which let us set a position in the array to a new value. Our main addition to this line of work is the "J-Tree Update Lemma" (Lemma 9), which gives a procedure that accomplishes this second task. The Update Lemma does not seem to have an analogue in any of the previously mentioned works.

# 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 1.** *We say a function $T : \mathbb{N} \to \mathbb{N}$ 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 2** (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 tape head, 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 two by the auxiliary tape head to the value $A_i$, where again $i$ is the integer whose binary representation lies to the left of the addressing tape head. 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 1.** *[GS89] 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 use $k$-tape Turing machines, for which we omit a formal definition as this 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 3** (Oracle Turing Machines)**.** *For a function $F : \{0,1\}^* \to \{0,1\}^*$, an $F$-oracle $k$-tape Turing machine has $k$ standard read/write "work tapes," in addition to a write-only 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 tape head prior to the oracle call. 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.*

*For a language $L$, we define an "$L$ oracle" to be an oracle for the characteristic function of $L$, which sends accepted inputs to 1 and rejected inputs to 0.*

The write-only nature of the oracle tapes is only relevant when working with 1-tape machines, where we don't want the machine to "cheat" and use it's oracle tapes as additional work tapes.

**Definition 4** (Time Bounded Classes)**.** *For a function $T : \mathbb{N} \to \mathbb{N}$, 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 5** (Time/Space Classes)**.** *For functions $T, S : \mathbb{N} \to \mathbb{N}$, 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 6** (Nondeterministic Time/Space Classes)**.** *For functions $T, S, G : \mathbb{N} \to \mathbb{N}$, 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 2.** *[GS89] 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. Finally, we formally define our measures of circuit complexity and $K^t$ complexity (time-bounded Kolmogorov complexity):

**Definition 7.** *Let $x \in \{0,1\}^N$. We say that $x$ "has circuits of size $s$" if there is a boolean circuit $C$ of fan-in 2 over the basis $\wedge, \vee, \neg$ on $\lceil \log N \rceil$ variables such that $C$ has at most $s$ gates and for all $0 \leq i < N$, $C(i)$ equals the $i^{th}$ bit of $x$ (where we convert between integers and binary strings in the standard way). We say that "$x$ has circuit complexity $s$" if $x$ does not have circuits of size $s-1$.*

**Definition 8.** *Fix some efficient universal Turing machine $U$. For $x \in \{0,1\}^*$, we define $K^t(x)$ to be the length of the smallest string $y$ such that $U$ outputs $x$ on input $y$ in at most $t$ steps.*

### 2.3 Search Problems

**Definition 9.** *A "search problem" is simply a relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$. The associated computational task is: given $x \in \{0,1\}^*$, find some $y \in \{0,1\}^*$ such that $(x,y) \in R$, or else determine that no such $y$ exists. We say that $R$ is "bounded" if there exists a polynomial $p$ such that $(x,y) \in R \Rightarrow |y| \leq p(|x|)$. We say that $R$ is "total" if $\forall x \in \{0,1\}^*$, $\exists y \in \{0,1\}^*$ such that $(x,y) \in R$.*

At some points in this work we will be concerned with functions that are efficiently computable with access to an oracle for solving some search problem. A subtlety arises here, since an instance of a search problem may have many valid solutions, and this may cause our oracle machine to have a variety of possible outputs. To formalize the notion of search problem oracles correctly, it seems necessary to define the following notion of "functional oracles":

**Definition 10.** *Let $R$ be a search problem. We say that $\mathcal{O} : \{0,1\}^* \to \{0,1\}^* \sqcup \{\perp\}$ is a "functional oracle" for $R$ if the following holds for all $x \in \{0,1\}^*$:*

1. *If there exists a $y$ such that $(x,y) \in R$ then $(x, \mathcal{O}(x)) \in R$.*
2. *If there does not exist a $y$ such that $(x,y) \in R$ then $\mathcal{O}(x) = \perp$*

Functional oracles give us a natural way to complexity classes solved by $R$-oracle machines for some bounded relation $R$ – an $R$-oracle machine computes a certain function, or solves some other search problem, if it has a valid output on every input when given access to an arbitrary functional oracle for $R$.

## 3 Pigeonhole Principles

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

### 3.1 Generators and Inverters

**Definition 11.** *Let $C : \{0,1\}^n \to \{0,1\}^m$, with $m > n$. We call such a map which extends its input length a "generator." The "seed length" of this generator is $n$, and its "output length" is $m$.*

By the "dual weak pigeonhole principle," if $m > n$, any function mapping $2^n$ "pigeons" to $2^m$ "holes" must leave some hole empty. We thus know there must exist a $y \in \{0,1\}^m$ such that $\forall x \in \{0,1\}^n$, $C(x) \neq y$. We call such a string $y$ an "empty pigeonhole" for the generator $C$.
The primary subject of [Kor21] was the problem EMPTY, where we are given a generator 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 generator and a supposed inverse of the generator, and must find a witness to the fact that a generator has no inverse. This was referred to by Jeřábek as the "retractive pigeonhole principle."[Jeř07].

**Definition 12.** *Let $C : \{0,1\}^n \to \{0,1\}^m, D : \{0,1\}^m \to \{0,1\}^n$, where $m > n$; we call such a $D$ an "inverter" for the generator $C$, and collectively we will refer to $C, D$ as a "generator/inverter pair" or "g.i. pair." By the "retractive pigeonhole principle," we know that there must exist some $y \in \{0,1\}^m$ such that $C(D(y)) \neq y$; we call such a $y$ "incompressible" with respect to the pair $C, D$.*

*We call $D$ a "proper inverter" for $C$ if $C, D$ satisfy the following for all $y \in \{0,1\}^m$: if there exists an $x \in \{0,1\}^n$ such that $C(x) = y$, then $C(D(y)) = y$. Note that when $D$ is a proper inverter for $C$, any incompressible string for the pair $C, D$ is necessarily an empty pigeonhole for $C$.*

*When a g.i.-pair has seed length $n$ and output length $2n$, we will say that it has "stretch 2."*

It should be noted that when $C$ is polynomial time computable, or is specified by some boolean circuit, there always exists a proper inverter for $C$ computable efficiently with an **NP** oracle, which simply finds the lexicographically first preimage of a string under $C$ or else outputs something arbitrary when none exist. In this sense, the work of [Kor21], which studies the complexity of EMPTY with respect to **NP**-oracle reductions, was essentially studying a special case of this problem, where the inverter is the "canonical proper inverter" which searches for the lexicographically first preimage.

For most of this work it will be convenient focus on the special case of functions which exactly double their input length (g.i. pairs of stretch 2). We thus utilize the following lemma, which tells us that if our goal is to find an incompressible string with respect to some g.i. pair, we can assume it has stretch 2 without loss of generality:

**Lemma 3.** *Let $C, D$ be a g.i. pair with seed length $n$ and output length $m$. Then there is a g.i. pair $C', D'$ of seed length $n$ and output 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 [Kor21] for the special case when $D$ is a proper inverter for $C$, but the same proof extends to the more general case stated here. □

In [Kor21], the problem EMPTY was studied as a search problem, where the generator $C$ 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 pair. However for our purposes it will be more suitable to study a *uniform* version of this problem, where we have a sequence of pairs $C_n, D_n$ of increasing size, and each are computable within some uniform complexity class.

**Definition 13.** *A uniform generator/inverter pair is a pair* $\mathcal{C} = \{C_n : \{0,1\}^{\ell(n)} \to \{0,1\}^{m(n)}\}_{n\in\mathbb{N}}$, $\mathcal{D} = \{D_n : \{0,1\}^{m(n)} \to \{0,1\}^{\ell(n)}\}_{n\in\mathbb{N}}$ *for some pair of functions* $m, l : \mathbb{N} \to \mathbb{N}$ *such that* $\ell(n) < m(n)$ *for all* $n$*, and* $m(n)$ *is bounded above by a polynomial in* $n$*.*

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

## 3.2 Particular G.I. Pairs of Interest

We now introduce g.i. pairs whose incompressible strings have certain desirable properties for which no explicit constructions are currently known.

### 3.2.1 Generators 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 [Kor21]:

**Lemma 4.** *For sufficiently large* $N \in \mathbb{N}$*, there is function* $\mathsf{f}_N : \{0,1\}^{N-1} \to \{0,1\}^N$ *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* $\mathsf{f}_N$*.*

We now define the search problem associated with finding preimages of strings under $\mathsf{f}_N$, which has commonly been referred to as the "circuit synthesis problem:"

**Definition 14** (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* $0 \le i < N$ *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" [KC00], which is not known to admit a search-to-decision reduction. By the same arguments underlying Lemma 4, 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 inverter for $\mathsf{f}_N$ computable in polynomial time with a Circuit Synthesis oracle, and thus any incompressible string for this pair is an $N$-bit string with high circuit complexity. As foreshadowed in Section 2.3, a subtly arises here since there are many valid functional oracles $\mathcal{O}$ for the search problem Circuit Synthesis, and for any such oracle, $\mathsf{f}_N, \mathcal{O}\,|_{\{0,1\}^N}$ will be a g.i. pair satisfying the required properties. However, an important observation is that if Circuit Synthesis is in **FP**, then there is a particular functional oracle $\mathcal{O}$ for this problem which is computed by some polynomial time algorithm, and in this case we have a fixed g.i. pair with the required properties where both the generator and inverter are computable in polynomial time.

We similarly have the following:

**Lemma 5.** *For any fixed polynomial* $p$*, there is a uniform g.i. pair whose generator is computable in polynomial time, and whose inverter 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 pair have* $K^{p(n)}$ *complexity* $\ge n-1$*.*

**Lemma 6.** *There is a uniform g.i. pair whose generator is computable in polynomial time, and whose inverter is computable with an oracle for* Formula Synthesis *(given a truth table find a short formula if one exists). The incompressible strings for this pair 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 g.i. pair exists whose generator is computable efficiently, whose inverter 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 uniform g.i. pair related to the large prime construction problem. This g.i. pair is unlike the previous ones, in that both the generator and inverter have the same complexity: each will require an oracle for factoring.

**Theorem 1.** *There is a generator* $\mathsf{P} : \{0,1\}^{n+\lceil \log n \rceil + 3} \to \{0,1\}^{n+\lceil \log n \rceil + 4}$ *and an inverter* $\mathsf{R}$ *for* $\mathsf{P}$*, each computable uniformly in polynomial time with a factoring oracle, such that given an incompressible string for* $\mathsf{P}, \mathsf{R}$*, a* $32n$*-bit prime of magnitude* $> 2^n$ *can be constructed in polynomial time with a factoring oracle.*

We can immediately note that we will not encounter the issue of dealing with different "functional oracles" here, since the factoring problem has a unique solution on all inputs. This theorem is an algorithmic analogue of a well known result of Paris Wilkie and Woods [PWW88], and our proof will follow from analysing the computational resources needed to carry out their construction. We start with the following useful lemma:

**Lemma 7.** *For sufficiently large* $n$ *there exists a pair of efficiently computable maps* $f : \{0,1\}^{n-2} \to \{0,1\}^n$*,* $g : \{0,1\}^n \to \{0,1\}^{n-2}$*, such that for all* $n$*-bit primes* $p$ *we have* $f(g(p)) = p$*.*

The constant 2 is arbitrary and can be replaced with any positive integer; we give a proof of this more general case in Appendix A. The encoding follows from the observation that the primes can be easily shown to have natural density $< \epsilon$ by considering their structure with respect to divisibility by the first $O(1)$ primes.

We will also use the following notation at various points in the proof:

**Definition 15.** *Let* $x \in \mathbb{N}$*, and let* $\prod_{i \leq v} p_i^{e_i}$ *be the prime factorization of* $x$ *where the* $p_i$ *are in increasing order of magnitude. We define* $I_x$ *to be the interval* $[\ell]$ *where* $\ell = \sum_{j \leq v} e_j \lceil \log p_j \rceil$*. We will think of the interval* $I_x$ *as being composed of* $e_1$ *intervals of length* $\lceil \log p_1 \rceil$*, followed by* $e_2$ *intervals of length* $\lceil \log p_2 \rceil$*, and so on. In this way, we see that every position in the interval* $I_x$ *determines a prime power* $p^k$ *dividing* $x$ *and an index* $i \in [\lceil \log p \rceil]$*, and conversely every such* $p^k, i$ *determines a position in* $I_x$*.*

We are now ready to prove Theorem 1:

*Proof of Theorem 1.* We will use the shorthand $N = 2^n$ for the remainder of the proof. Consider an arbitrary string $xs \in \{0,1\}^{n+\lceil \log n \rceil + 4}$, where $|x| = n$ and $|s| = \lceil \log n \rceil + 4$. We will consider $x$ as an integer in the range $[N]$, and $s$ as representing a position in the interval $I_{N^{16}+x}$. Since $N^{32} + x > N^{32}$, we see that $I_{N^{32}+x}$ has length at least $32n$, and thus any $\lceil \log n \rceil + 4$ bit string can

be identified uniquely with some position in $I_{N^{32}+x}$. As described above, this position is in turn fully defined by a prime power $p^k$ dividing $N^{32}+x$, together with an index $i \in [\lceil \log p \rceil]$. Further, it is clear that from $x$ and $s$ we can determine $p, k, i$ in $poly(n)$ time with a factoring oracle.

For some prime power $p^m$ and $y \in [N]$, we will say that $p^m \sim y$ if the following holds: $p^{m+1}$ has no multiple in the interval $[N^{32}+1, N^{32}+N]$, and $N^{32}+y$ is the smallest integer in the interval $[N^{32}+1, N^{32}+N]$ which is divisible by $p^m$. Clearly it can be tested in $poly(n)$ time whether $p^m \sim y$ using division. We can also see that for any prime $p \leq N$, there exists a unique $y \in [N]$ such that $p^m \sim y$ for some $m$, and such a $y$ can be computed from $p$ efficiently by testing increasing powers of $p$.

Now, our map $\mathsf{R}$ on input $xs$ will start by computing the values $p, k, i$ determined by $s$ and $I_{N^{32}+x}$ as described above. During this computation, $\mathsf{R}$ will test if each prime in the decomposition of $N^{32}+x$ is of magnitude $\leq N$. If one of them is not, $\mathsf{R}$ will "fail" and simply output an arbitrary string (say $1^{n+\lceil \log n \rceil+3}$ for concreteness). Otherwise, $\mathsf{R}$ next tests if $p^k \sim x$. If this holds, we have that $x$ can be recovered from $p$ efficiently, and in this case $\mathsf{R}$ maps $xs$ to $0ps$ where the prime $p$ is encoded in $n-2$ bits according to Lemma 7. Overall this can be done in the required $n+\lceil \log n \rceil+3$ bits. Now say that $p^k \sim x$ does not hold. In this case, $\mathsf{R}$ computes the unique integer $y \in [N]$ such that $p^m \sim y$ holds for some $m$. We then set $z = |x-y|$, and $b = sign(x-y) \in \{0,1\}$. By definition of $m$ we see that $k \leq m$, since $p^k$ does have a multiple in $[N^{32}+1, N^{32}+N]$, namely $x$. In addition we have that $p^m$ divides $N^{32}+y$ and $p^k$ divides $N^{32}+x$. So overall we conclude that $p^k$ must divide $|(N^{32}+y)-(N^{32}+x)| = z$. Now, consider the interval $I_z$, which has length at most $2n$ since $z \leq N$. Since $p^k$ divides $z$, there is some position in $I_z$ which determines $p, k, i$, and this position can in turn be encoded in $\lceil \log n \rceil + 1$ bits, call this encoding $s'$. Now $\mathsf{R}$ outputs $1zbs'$. Again we see that this output has length $n+\lceil \log n \rceil+3$ as required.

We now define the map $\mathsf{P}$, which simply unpacks the above encoding of an input $xs$ whenever $\mathsf{R}$ does not fail. When $\mathsf{P}$ sees an input of the form $0ps$ (where $p$ is encoded in $n-2$ bits as per Lemma 7), it recovers the unique $x$ such that $p^m \sim x$ for some $m$, and then outputs $xs$. Otherwise, if it sees an input of the form $1zbs'$, it computes the prime factorization $z$ to determine the values $p, k, i$ encoded by the position of $s'$ in $I_z$. From $p$ it then computes the unique $y \in [N]$ and $m$ such that $p^m \sim y$, and sets $x = y + (-1)^b z$. Finally, it computes the factorization of $N^{32}+x$, and determines the position in $I_{N^{32}+x}$ representing $p, k, i$, call it $j$; if $j \leq 2^{\lceil \log n \rceil+4}$, it then encodes the position $j$ as a $\lceil \log n \rceil + 4$ bit string and outputs $xs$; otherwise it outputs something arbitrary ($1^{n+\lceil \log n \rceil+4}$ for example).

By construction we see that whenever $\mathsf{R}$ does not "fail" on input $xs$, it outputs some value from which $\mathsf{P}$ then recovers $xs$. But the only way that $\mathsf{R}$ can fail on an input $xs$ is when the prime factorization of $N^{32}+x$ contains a prime $> N$. Thus if $xs$ is an incompressible string for $\mathsf{P}, \mathsf{R}$, $N^{32}+x$ must have prime factor $> N$, which we can compute given $x$ using a factoring oracle. □

# 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 g.i. pair $C, D$, 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 pair $C, D$, then it is hard to find a sequence of updates/queries which causes our data structure to fail.

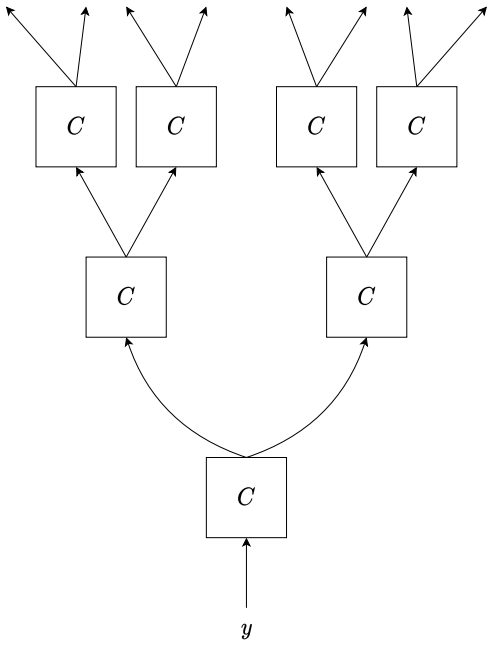


Figure 2: A J-tree of depth 3. Each arrow consists of $n$ wires, where $n$ is the seed length of the generator $C$.

**Definition 16** (J-trees)**.** *Let $C : \{0,1\}^n \to \{0,1\}^{2n}$ be a generator of seed length $n$ and stretch 2. We define $C_0, C_1 : \{0,1\}^n \to \{0,1\}^n$ to be the maps obtained by computing $C$ and taking the first $n$ and last $n$ bits of output respectively. Now, for any binary string $x \in \{0,1\}^*$, we define $C_x : \{0,1\}^n \to \{0,1\}^n$ as follows. When $|x| = 0$, $C_x$ is the identity, and when $|x| = 1$, $C_0, C_1$ are defined as above. In the general case, when $x = b_1 \cdots b_k$ for some $b_1, \ldots, b_k \in \{0,1\}$, $C_x$ is defined as:*

$$C_x = C_{b_k} \circ \cdots \circ C_{b_1}$$

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

$$C[y](x) = C_x(y)$$

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

*We will refer to $C^k[y]$ as a "J-tree," $C$ as its generator, $y$ 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 $C^k[y]$ where $k = 3$. For a given $C$ and depth $k$, we can construct a tree-like circuit which starts with one copy of $C$, then feeds each of its two $n$-bit output blocks into another copy of $C$, 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. $C^k[y]$ 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 $C^k[y]$ 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 $C^k[y](i)$. The

state of the data structure is described purely by $y$ and $C$, 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 generator $C$ $O(k)$ times, the update operation will require evaluating $D$, 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 $C$, that the resulting J-tree is in fact the true updated version of the original. We begin with the query or "access" operation:

**Lemma 8** (J-tree Access Lemma)**.** *Let $C^k[y]$ be a J-tree. Then for any $i \in \{0,1\}^k$ we can compute $C^k[y](i)$ in time $O(nk)$ given $i, y$ and using $O(k)$ evaluations of $C$.*

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

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

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

**Definition 17** (J-tree Modifications)**.** *Let $C^k[y]$, $C^k[y']$ be J-trees of depth $k$ and seed length $n$. We say that the relation* $\mathsf{Modified}(y, y', i, s, C)$ *holds if:*

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

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

**Lemma 9** (J-tree Update Lemma)**.** *There exist algorithms* Find *and* Verify *satisfying the following.*

Verify *takes as input a generator $C$ (specified as an oracle) of seed 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 $C$, and satisfies the property that if* $\mathsf{Verify}(y, y', i, s, C)$ *accepts, then* $\mathsf{Modified}(y, y', i, s, C)$ *must hold.*

Find *takes as input a g.i.-pair $C, D$ of seed 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 $\mathsf{FAIL}\langle e \rangle$, 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. $\mathsf{Find}(y, i, s, C, D)$ *outputs a string $y'$ such that* $\mathsf{Verify}(y, y', i, s, C)$ *accepts.*
2. $\mathsf{Find}(y, i, s, C, D)$ *outputs* $\mathsf{FAIL}\langle e \rangle$*, 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

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

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

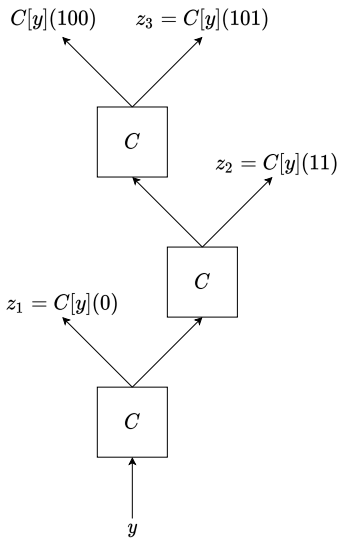


Figure 3: "Forward phase" of $\mathsf{Find}(y, 100, s, C, D)$ (where $k = 3$), during which the $z_1, \ldots, z_k$ are computed.

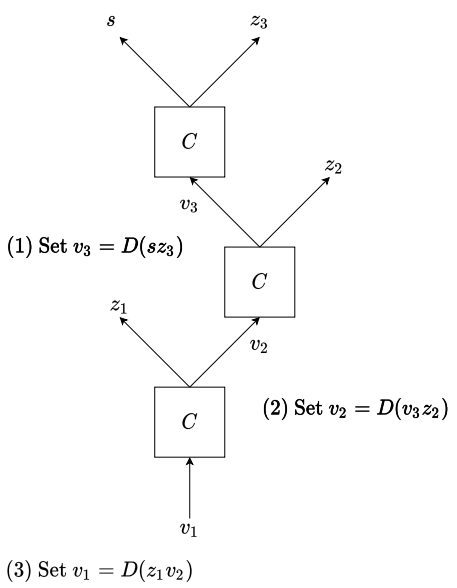


Figure 4: "Backward phase" of $\mathsf{Find}(y, 100, s, C, D)$ (where $k = 3$), during which the $v_1, \ldots, v_k$ are computed.

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:

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

We then check that $C(D(sz_k)) = sz_k$ (resp. $C(D(z_k s)) = z_k s$). 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:

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

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

We now show that if $\mathsf{Find}(y, i, s, C, D)$ succeeds and returns a string $y'$, then $\mathsf{Verify}(y, y', i, s, C)$ 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 $j < k$, $C_{b_j}(v_j) = v_{j+1}$ and $C_{\neg b_j}(v_j) = z_j$, and it returns $v_1$ as its output. So then we have that $C[v_1](b_1, \ldots, b_{j-1}, \neg b_j) = C[v_j](\neg b_j) = z_j$ for all $j \in [k]$. Further, we have that $C[v_1](b_1, \ldots, b_{k-1}, b_k) = C[v_k](b_k) = s$. So overall $\mathsf{Verify}(y, v_1, i, s, C)$ must accept if $\mathsf{Find}(y, i, s, C, D)$ succeeds and returns $v_1$.

It remains only to show that if $\mathsf{Verify}(y, y', i, s, C)$ accepts, then $\mathsf{Modified}(y, y', i, s, C)$ must hold. Recall that $\mathsf{Modified}(y, y', i, s, C)$ asserts that the two J-trees $C^k[y]$, $C^k[y']$ agree an all indices $i' \neq i$, and that $C^k[y'](i) = s$. If $\mathsf{Verify}$ accepts then this second condition holds trivially, since $\mathsf{Verify}$ explicitly checks that $C^k[y'](i) = s$ 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 $\mathsf{Verify}$ accepted, we have that

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

But by definition we also know that

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

and similarly

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

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

Next, we prove 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 10.** *(J-Tree Initialization Lemma) There is an algorithm* $\mathsf{Initialize}$ *which takes as input a g.i.-pair $C, D$ of seed 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 $i \in \{0,1\}^k$, $C^k[y](i) = s$, 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 $D$.*

*Proof.* We start by setting $v_1 = D(ss)$. When then check if $C(v_1) = ss$ and if not we return $\mathsf{Fail}\langle ss \rangle$. Otherwise we continue for each value $j \in [k]$ in increasing order, setting $v_j = D(v_{j-1}v_{j-1})$ and checking that $D$ finds a valid preimage at each step and returning an incompressible string if it does not. We then return $v_k$ if all checks pass.

Say $\mathsf{Find}(s, k, C, D)$ succeeds and returns $v_k$. By induction we show that $C[v_j](i) = s$ for all $i \in \{0,1\}^j$. For the base case, since $C(v_1) = ss$ have that $C[v_1](i) = s$ for all $i \in \{0,1\}$. For the inductive case say $i = bi'$ for $b \in \{0,1\}, i' \in \{0,1\}^{j-1}$. So then we have that $C[v_j](i) = C[C_b(v_j)](i')$. By assumption that $\mathsf{Find}(s, C, D)$ succeeds we have that $C_b(v_j) = v_{j-1}$, and by the inductive hypothesis we have that $C[v_{j-1}](i') = s$, completing the proof. □

In the proofs to come it will also be useful to have the following "compression" lemma. This Lemma is implicit in the proof of the main theorem in [Kor21].

**Lemma 11** (J-Tree Static Compression Lemma)**.** *Let $C, D$ be a g.i.-pair of seed 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* $\mathsf{Compress}$ *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 $C^{\lceil \log \ell \rceil}[y](i) = s_i$ 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 $C, D$.*

*Proof.* This follows from repeated application of the Find procedure in Lemma 9. We initialize $y = 0^n$. Now, for each $i \in [\ell]$, we perform $\mathsf{Find}(y, i, s_i, C, D)$, and if Find fails and an incompressible string $e$, then Compress fails and outputs $e$. Otherwise we set $y = \mathsf{Find}(y, i, s_i, C, D)$. We then repeat for the next value of $i$. If we succeed for all $i$, we output the final value of $y$. Correctness and runtime of this algorithm follow from Lemma 9. □

# 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 g.i. pair. We then show that either this simulation is successful, or there is an explicit construction algorithm that prints incompressible strings for this g.i. pair.

## 5.1 Proofs of Main Theorems

**Theorem 2.** *Let $\mathcal{C}, \mathcal{D} = \{C_n, D_n\}_{n \in \mathbb{N}}$ be a uniform g.i. pair.*
*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 g.i. pair. By Lemma 3, 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 this 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 pair. By definition of a uniform g.i. pair, we see that the seed length of $C, D$ will by $poly(2^{\epsilon n}) = T(n)^{O(\epsilon)}$, and by assumption its output length is twice its seed length. We will use $W$ to denote its seed length for the remainder of the proof.

We start by invoking Lemma 1, 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 $\mathtt{Mem} \in \{0,1\}^W$ which will be used as a seed in a J-tree with

generator $C$ 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 $C^k[\texttt{Mem}](i) = b^W$ 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).

At the start of the first phase, $\mathcal{S}_\epsilon$ initializes $\texttt{Mem}$ to a value such that $C^k[\texttt{Mem}](i) = 0^W$ 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 10, this can be accomplished in time $O(kW) = T^{O(\epsilon)}$ with oracle access to $C, D$. 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 tape head 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 $\mathsf{Load}$, we compute the first bit of $C^k[\texttt{Mem}](i)$ and update the auxiliary tape at its current tape head position to this value. If the operation is an $\mathsf{Update}$, 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 $\texttt{Mem}$ to the value returned by this $\mathsf{Find}$ 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 $\mathsf{Find}/\mathsf{Initialize}$ operations and the evaluations of $C^k[\texttt{Mem}]$ can be carried out in $poly(W)$ space, since in particular they require at most $poly(W)$ time by Lemmas 8, 9, and 10. 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 $\mathsf{Find}$ Never Fails Then the Simulator Works**

We now show that if $\mathcal{S}_\epsilon$ completes its computation on an input $x$ without any $\mathsf{Initialize}$ or $\mathsf{Find}$ operation failing, then $\mathcal{S}_\epsilon$ accepts $x$ if and only if $x \in L$. To prove this, we show by induction on

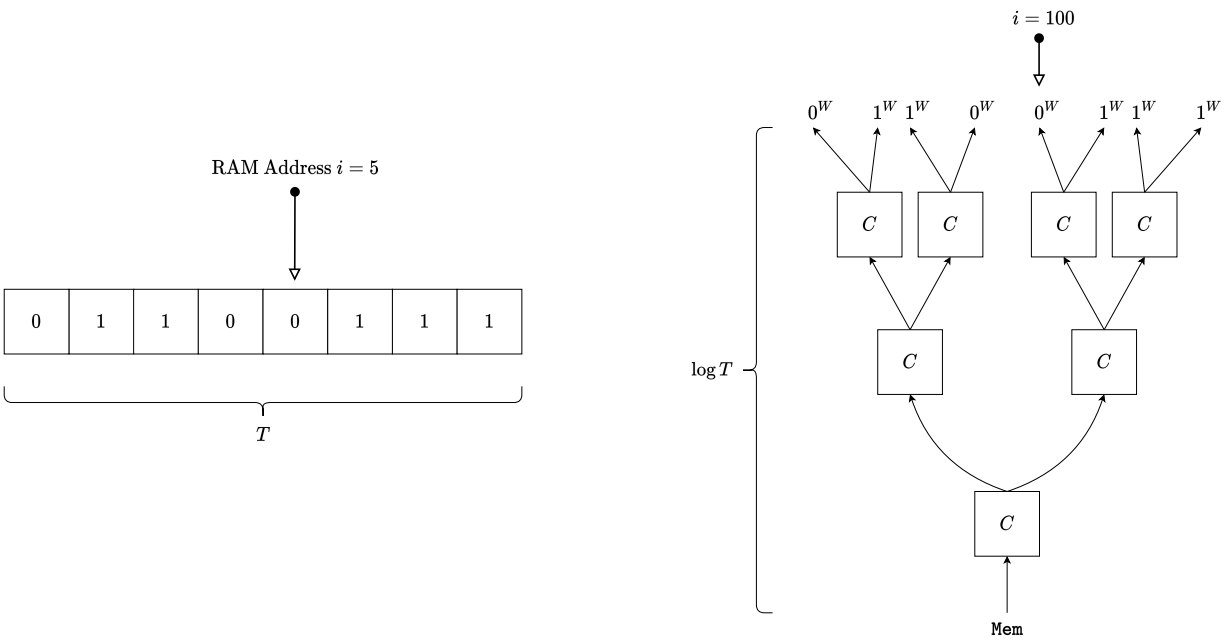


Figure 5: Simulating a RAM memory with a J-tree. While the RAM memory on the left requires $T$ bits to store explicitly, the virtual RAM on the right can be stored with $|\texttt{Mem}| = W = T^{O(\epsilon)}$ bits.

$t \in [T]$ 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 9, 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 generators $C_{\lceil 2^{\epsilon n} \rceil}, D_{\lceil 2^{\epsilon n} \rceil}$ 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 generators $C_m, D_m$ 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 $L \in \mathbf{RAM\text{-}TIME}[T(n)]$ and every $\epsilon > 0$, there is a machine $S_\epsilon$ 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 $\epsilon$ as we take $\epsilon$ to zero. This yields the stated theorem. □

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

**Theorem 3.** *Let $T$ be an exponential time bound, and let $\mathcal{C}, \mathcal{D}$ be a uniform g.i. pair. 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 $L \in \mathbf{RAM\text{-}TIME}[T(n)]$ and every $\epsilon > 0$, there is nondeterministic 1-tape Turing machine with oracle access to $\mathcal{C}$ 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 $\mathbf{RAM\text{-}TIME}[T(n)]$, 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 9. However, recall that Lemma 9 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 $\mathcal{C}$ 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 $\mathcal{C}$ oracle. This immediately gives us a low space nondeterministic simulator $\mathcal{S}_\epsilon$ with similar properties as that in the proof of Theorem 2.

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. At the beginning of $\mathcal{S}_\epsilon$'s simulation, it guesses a single seed for a separate J-tree of the same seed length, call this seed $\mathtt{Up}$. 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 $\mathtt{Mem}' = C^k[\mathtt{Up}](t)$, and then check if $\mathsf{Verify}(\mathtt{Mem}, \mathtt{Mem}', i, s^W, C)$ holds. If so we then set $\mathtt{Mem} = \mathtt{Mem}'$ and continue to the next phase, 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 2, 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 2.

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 2 (corresponding to $L, \epsilon$) 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 $\mathtt{Mem}_1, \ldots, \mathtt{Mem}_T \in \{0,1\}^W$ (where $W$ is the seed length used by the simulation), such that $\mathtt{Mem}_{t+1}$ represents a correct update to the RAM memory previously represented by $\mathtt{Mem}_t$ after time step $t$. Now, we use the Compress procedure to construct a seed $\mathtt{Up} \in \{0,1\}^W$ such that for all $t \in \{0,1\}^k$, $C^k[\mathtt{Up}](t) = \mathtt{Mem}_t$. By Lemma 11, in $poly(T(n))$ time we can either find such a seed $\mathtt{Up}$, or else find an incompressible string for our g.i. pair. But if such a string $\mathtt{Up}$ 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 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 4.** *Let $T$ be an exponential time bound, and let $\mathcal{C}, \mathcal{D}$ be a uniform g.i. pair. 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 $\mathcal{C}$ 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 $\mathtt{Up}$ which encodes a sequence of RAM-seed updates, our new simulator $\mathcal{S}_\epsilon$ also guesses a second seed $\mathtt{Wit}$ 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 $C^k[\mathtt{Wit}](t)$, 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 seed length of the generator 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 $\mathtt{Up}, \mathtt{Wit} \in \{0,1\}^W$ such that $\mathtt{Wit}$ encodes $z$ (as described above), and $\mathtt{Up}$ 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 $\mathtt{Mem}_1, \ldots, \mathtt{Mem}_T$ for this computation. We then proceed as in the previous Theorem, attempting to compress the $\mathtt{Mem}_t$ to a single seed $\mathtt{Up}$ and the bits of $z$ to another seed $\mathtt{Wit}$, both using the Compress procedure. By assumption that our main nondeterministic simulation rejected $x$, if we get to this point we know that one of these 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:*

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

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

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

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

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

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 exponential 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. In any case, the results of the previous section give us the following:

**Theorem 5.** *If Hypothesis 1 holds, then for any uniform g.i. pair $\mathcal{C}, \mathcal{D}$ 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 $n$.*

*Proof.* This follows from Theorem 2; since the $\mathcal{C}$ and $\mathcal{D}$ oracle calls are of length $T(n)^{O(\epsilon)}$, if both $\mathcal{C}$ and $\mathcal{D}$ are computable in polynomial time then the oracles can be abandoned without effecting the resource bounds of the simulation. □

For Hypothesis 2, we have:

**Theorem 6.** *If Hypothesis 2 holds, then for any uniform generator $\mathcal{C}$ computable in polynomial time, there exists there is a polynomial time* **NP***-oracle algorithm which, given $1^n$, prints an empty pigeonhole for $C_n$ for infinitely many $n$. In particular, $\mathbf{E}^{\mathbf{NP}} \not\subseteq \mathbf{size}[2^n/2n]$.*

*Proof.* This follows from Theorem 4. As noted in Section 3, any polynomial time generator $\mathcal{C}$ has a canonical "proper inverter" $\mathcal{D}$ computable with an **NP** oracle, such that the incompressible strings for this pair are empty pigeonholes for $\mathcal{C}$. The theorem then follows from the above proof, where we can abandon the $\mathcal{C}$ oracle in our simulation if $\mathcal{C}$ is computable in polynomial time since the oracle calls we make are small. □

Finally, the strongest hypothesis gives us the following:

**Theorem 7.** *If Hypothesis 3 holds, then for any uniform g.i. pair $\mathcal{C}, \mathcal{D}$ where $\mathcal{C}$ is computable in polynomial time, there is a polynomial time algorithm using a $\mathcal{D}$ oracle which, given $1^n$, prints an incompressible string for $C_n, D_n$ for infinitely many $n$.*

*Proof.* This follows from Theorem 3, where again we use the fact that $\mathcal{C}$ is computable in polynomial time to eliminate the $\mathcal{C}$ oracle. □

To illustrate the use of these theorems, we first consider their implications for the generators 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 1.** *For each of the above mentioned g.i. pairs 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.*

*Proof.* The first and second implications follow directly from the above theorems. For the last implication, there is a slight subtlety related to the fact that the compression problem can generally have many valid solutions per instance, and so an oracle for this problem must be defined according to our notion of "functional oracles" (defined in Section 2.3). Thus, for a particular compression problem we have the following dichotomy for every one of its function oracles $\mathcal{O}$: either the g.i. pair defined by $\mathcal{O}$ returns solutions that can be used as witnesses in the low space simulation for all but finitely many inputs, or else there is an explicit construction algorithm with access to $\mathcal{O}$ which prints strings of high complexity. So assuming Hypothesis 3, every $\mathcal{O}$ must fail to help the simulation for infinitely many inputs. Thus, given access to an arbitrary functional oracle for

the compression problem, the explicit construction algorithm will work for infinitely many input lengths, although the choice of functional oracle might affect which infinite set of input lengths the algorithm works on. □

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 Appendix B 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 generator and inverter require a factoring oracle. In addition, the second implication above is trivial when applied to the construction of large primes, since primality testing is in **P** [AKS02]. However we still have the following:

**Corollary 2.** *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* $n$*).*

# 6 Acknowledgements

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

# References

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in P. *Ann. of Math*, 2:781–793, 2002.

[BFL90] L. Babai, L. Fortnow, and C. Lund. Nondeterministic exponential time has two-prover interactive protocols. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 16–25 vol.1, 1990.

[FLvMV05] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. ACM*, 52(6):835–865, nov 2005.

[For00] Lance Fortnow. Time–space tradeoffs for satisfiability. *Journal of Computer and System Sciences*, 60(2):337–353, 2000.

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

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Proceedings of the Symposium on Logical Foundations of Computer Science: Logic at Botik '89*, page 108–118, Berlin, Heidelberg, 1989. Springer-Verlag.

[IKW02] 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.

[Jeř07] Emil Jeřábek. On independence of variants of the weak pigeonhole principle. *Journal of Logic and Computation*, 17(3):587–604, 2007.

[Kab01] Valentine Kabanets. Easiness assumptions and hardness tests: Trading time for zero error. *Journal of Computer and System Sciences*, 63(2):236–252, 2001.

[KC00] Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 73–79, New York, NY, USA, 2000. Association for Computing Machinery.

[KL80] Richard M. Karp and Richard J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, page 302–309, New York, NY, USA, 1980. Association for Computing Machinery.

[Kor21] Oliver Korten. The hardest explicit construction. In *62nd Annual Symposium on Foundations of Computer Science*, 2021.

[Maa84] 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, page 401–408, New York, NY, USA, 1984. Association for Computing Machinery.

[Mer88] 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.

[OS17] 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, page 665–677, New York, NY, USA, 2017. Association for Computing Machinery.

[Pap94] 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.

[PWW88] 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.

[Rob92] 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.

[Wil05] Ryan Williams. Inductive time-space lower bounds for sat and related problems. *Computational Complexity*, 15:433–470, 2005.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 1982.

# A Succinct Representation of Primes

We give here a proof of a generalization of Lemma 7.

**Lemma.** *Let $c \in \mathbb{N}$. For sufficiently large $n$ there exists a pair of polynomial time computable maps $f : \{0,1\}^{n-c} \to \{0,1\}^n$, $g : \{0,1\}^n \to \{0,1\}^{n-c}$, such that for all $n$-bit primes $p$ we have $f(g(p)) = p$.*

*Proof.* Let $p_1, \ldots, p_k$ be the first $k$ primes, and consider the probability that a uniformly random tuple in $\mathbb{Z}/p_1\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k\mathbb{Z}$ has no zero-entry. This probability is:

$$\prod_{j=1}^{k} \frac{p_j - 1}{p_j}$$

We see that this probability approaches zero for large $k$, so we now fix some $k$ such that this probability is at most $2^{-2c}$. Let $M = \prod_{j \leq k} p_k$. By the Chinese remainder theorem, this implies the existence of a set $S \subseteq [M]$ of size at most $2^{-2c}M$, such that any integer $x$ which is not divisible by any of $p_1, \ldots, p_k$ can only be congruent to a value in $S$ modulo $\mathbb{Z}/M\mathbb{Z}$. So in particular, any prime $p > p_k$ can only take on this small set of values modulo $M$. This immediately gives the required encoding of $n$ bit primes larger then $p_k$ for sufficiently large $n$: we simply divide by $M$ which saves $\lfloor \log M \rfloor$ bits, and encode the remainder using $\lceil \log(2^{-2c}M) \rceil \leq \lfloor \log M \rfloor - 2c + 1$ bits, so overall we can use at most $n - c - 1$ bits to encode an $n$ bit prime greater then $p_k$, provided $n >> M$ (recall that $M$ is a fixed constant). This encoding is not quite good enough, as it does not work for the primes $p_1, \ldots, p_k$. However by adding a single bit to the encoding scheme we can trivially extend to it encode these $k = O(1)$ values, completing the proof. □

## B Alternate Proof of Nondeterministic Low-Space Simulation

We give here an alternative proof of the following theorem using the well-known "Easy Witness Lemma:"

**Theorem.** *If, for some exponential time bound $T$ and $\epsilon > 0$ we have:*

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

*then* $\mathbf{E}^{\mathbf{NP}} \not\subseteq \mathbf{size}[2^n/2n]$.

We start by stating the "Easy Witness Lemma" for $\mathbf{E}^{\mathbf{NP}}$:

**Lemma 12.** *[IKW02] If $\mathbf{E}^{\mathbf{NP}} \subseteq \mathbf{size}[f(n)]$, then for any exponential time nondeterministic turing machine $\mathcal{M}$ and every input $x$ that $\mathcal{M}$ accepts, there is a circuit of size $O(f(|x|)^k)$ (for some fixed universal constant $k$) whose truth table is a sequence of nondeterministic bits causing $\mathcal{M}$ to accept $x$.*

The only other tool we need is the **NP**-completeness of 3-SAT under hyper-efficient reductions due to [FLvMV05], which can be readily used to show the following:

**Theorem 8.** *Let $L \in \mathbf{NTIME}[T(n)]$. There is a nondeterministic machine $\mathcal{M}$ deciding $L$ in time $T'(n) = \tilde{O}(T(n))$ of the following special form. $\mathcal{M}$ guesses $T'(n)$ nondeterministic bits on a separate read-only random-access "guess tape" at the beggining of its computation in one step. $\mathcal{M}$ then runs in time $T'(n)$ and uses space at most $\tilde{O}(\log T(n) + n)$ on its work tapes and accepts or rejects. As usual, we say $\mathcal{M}$ accepts an input $x$ if there is some initial guess tape configuration causing it to accept, and otherwise we say that it rejects $x$. We do not count the guess tape towards the space usage of $\mathcal{M}$.*

*Proof.* The stated simulator simply iterates over each clause of a 3-SAT instance produced by applying the efficient 3-SAT reduction from $L$. For each clause, the verifier confirms that the clause is satisfied by checking the variable assignments on the random access guess tape. Due to the efficiency/high degree of uniformity in the 3-SAT reduction of [FLvMV05], it is straightforward to confirm that this verifier runs in the stated time and space bounds. □

Combining this with the easy witness lemmas we get:

**Theorem 9.** *There is a universal constant $k$ such that if $\mathbf{E}^{\mathbf{NP}} \subseteq \mathbf{size}[f(n)]$, then for all exponential time bounds $T(n)$ we have:*

$$\mathbf{NTIME}[T(n)] \subseteq \mathbf{NTISPG}[T(n)f(n)^k, f(n)^k, f(n)^k]$$

*Proof.* Let $L$ be in $\mathbf{TIME}[T(n)]$, and let $\mathcal{M}$ be the special machine deciding $L$ which is given by Theorem 8. Note that although this machine is defined in a nonstandard way (guessing $\tilde{O}(T(n))$ bits in one step on a random access tape), it can be simulated by a standard multi-tape nondeterministic machine $\mathcal{M}'$ in $poly(T(n)) = 2^{O(n)}$ time in such a way that there is a one-to-one correspondence between the nondeterministic guesses causing $\mathcal{M}$ and $\mathcal{M}'$ to accept on any given input $x \in L$. In particular, the machine $\mathcal{M}'$ simply guesses $\tilde{O}(T(n))$ bits one at a time and stores them on one of its tapes, and then simulates random access to these bits during the verification by walking along the tape, incurring a time overhead of $\tilde{O}(T(n))$. Now by the $\Delta_2$ Easy Witness Lemma, assuming $\mathbf{E}^{\mathbf{NP}} \in \mathbf{size}[f(n)]$ we have that for any $x \in L$, there is a circuit of size $f(n)^k$ (for a fixed universal

constant $k$) whose truth table is a sequence of guesses causing $\mathcal{M}'$ to accept $x$, and by the above relation between $\mathcal{M}$ and $\mathcal{M}'$ this truth table must also cause $\mathcal{M}$ to accept $x$.

We can then simulate $\mathcal{M}$ in low space and low nondeterminism as follows: at the start, instead of guessing a full random access tape of nondeterministic bits, we guess a circuit $C$ of size $f(n)^k$ encoding a potential witness, which requires $\tilde{O}(f(n))$ nondeterministic bits. We then run $\mathcal{M}$'s random access verification procedure, replacing unit cost accesses to its random access guess tape with evalutations of the circuit $C$. The time overhead to simulate each step of $\mathcal{M}$'s verification is at most $poly(f(n)^k)$ (to evaluate $C$ on a requested index), and the additional space usage is $\tilde{O}(f(n))$ to store $C$ (on top of the original space usage of the verifier which is $\tilde{O}(\log T(n) + n)$). Since $T$ is an exponential time bound and $f(n) \geq n$, overall this simulation runs in time $T(n)poly(f(n))$ and uses space and nondeterminism at most $poly(f(n))$, and by the easy witness lemma this simulation must correctly decide the language $L$. □

The main theorem is then obtained by combining the above with the following amplification lemma for $\mathbf{E}^{\mathbf{NP}}$:

**Lemma 13.** *If* $\mathbf{E}^{\mathbf{NP}} \subseteq \mathbf{size}[2^n/2n]$, *then* $\mathbf{E}^{\mathbf{NP}} \subseteq \bigcap_{\epsilon>0} \mathbf{size}[2^{\epsilon n}]$.

This was essentially proven in [Kor21], for the case when both inclusions are of the "infinitely often" type. We give a proof of this version below for completeness.

*Proof.* We prove this statement by contrapositive. Say there exists a language $L \in \mathbf{E}^{\mathbf{NP}}$ and a constant $\epsilon > 0$ such that for infinitely many $n \in \mathbb{N}$, $L^n$ has circuit complexity $2^{\epsilon n}$. By assumption $L$ can be computed in time $2^{cn}$ with an **NP** oracle for some fixed $c \in \mathbb{N}$. Thus, given $1^n$, we can also compute the truth table of $L^n$, which has length $2^n$, in time $2^n 2^{cn} = 2^{(c+1)n}$ with an **NP** oracle.

We now define a language $H \in \mathbf{E}^{\mathbf{NP}}$ such that $H \notin \mathbf{size}[2^n/2n]$. To compute $H$ on inputs of length $n$, we start by constructing a circuit $C$ with the following properties:

1. $C$ has $W = 2^n - 1$ inputs and $2W$ outputs, and has size $O(W^3) = O(2^{3n})$.
2. Given a string outside the range of $C$, we can find an $2^n$-bit truth table which cannot be computed by a circuit of size $2^n/2n$ in $poly(2^n)$ time with an **NP** oracle.

By [Kor21] such a $C$ can be constructed in time $poly(2^n)$; this can also be seen by combining Lemmas 3 and 4. Now, for each $m \in \{\lceil \frac{6}{\epsilon} \rceil n, \ldots, \lceil \frac{6}{\epsilon} \rceil (n+1)\}$, we do the following:

1. Compute the truth table of $L^m$, which takes time $2^{(c+1)m}$ with an **NP** oracle.
2. Attempt to find a seed $y \in \{0,1\}^W$ such that for all $i \in \{0,1\}^m$, $C^m[y](i) = b_i^W$ where $b_i$ denotes the $i^{th}$ bit of $L^m$. This can be accomplished using the Compress algorithm. By Lemma 11, in $poly(|C|m) = O(2^{3n}n)$ time this procedure will either find such a $y$, or else find an empty pigeonhole $e$ for $C$. By construction of $C$, if we can find one of its empty pigeonholes this in turn allows us to construct a truth table of length $2^n$ and circuit complexity exceeding $2^n/2n$, also in $poly(2^n)$ time. If this happens, we halt the entire procedure and use this truth table to decide whether to accept or reject the input.

If we get through all values of $m$ without finding an empty pigeonhole, we reject the input. This concludes the definition of the machine deciding $H$. By construction we have $H \in \mathbf{E}^{\mathbf{NP}}$.

By definition, for a particular input lenght $n$, if there is some $m \in \{\lceil \frac{6}{\epsilon} \rceil n, \ldots, \lceil \frac{6}{\epsilon} \rceil (n+1)\}$ for which the above procedure finds an empty pigeonhole for $C$, then $H^n$ will have circuit complexity at least $2^n/2n$. Now, say this does not hold for a particular $n$. Then we claim that for all input lengths

$m \in \{\lceil \frac{6}{\epsilon} \rceil n, \ldots, \lceil \frac{6}{\epsilon} \rceil (n+1)\}$, $L^m$ has circuit complexity less then $2^{\epsilon m}$, provided $n$ is sufficiently large. This follows directly from Lemma 8: if there is some $y \in \{0,1\}^W$ such that the first bit of $C^m[y](i)$ equals the $i^{th}$ bit of $L^m$ for all $i \in \{0,1\}^m$, then this directly gives a circuit of size $O(|C|m)$ allowing us to compute $L^m$ using $y$ as advice (namely the algorithm described in Lemma 8, which can be made non-uniform at no additional cost by [Kor21]). For $n$ sufficiently large, this $O(|C|m)$ term will be strictly less then $2^{\epsilon m}$. Thus, if $H \in \mathbf{size}[2^n/2n]$, then for all sufficiently large $n$, $L^m$ has circuit complexity less than $2^{\epsilon m}$ for all $m \in \{\lceil \frac{6}{\epsilon} \rceil n, \ldots, \lceil \frac{6}{\epsilon} \rceil (n+1)\}$. But since the intervals $\{[\lceil \frac{6}{\epsilon} \rceil n, \lceil \frac{6}{\epsilon} \rceil (n+1)]\}_{n \in \mathbb{N}}$ cover all but finitely many natural numbers, this implies $L^m$ has circuit complexity less than $2^{\epsilon m}$ for all sufficiently large $m$, contradicting our hardness assumption for $L$. So $H \notin \mathbf{size}[2^n/2n]$, completing the proof. □



ECCC ISSN 1433-8092
https://eccc.weizmann.ac.il