Discrete Mathematics and Theoretical Computer Science . . . . . .
. . . . . **vol. 27:3 #15 (2025)**

P *has polynomial-time finite-state verifiers*[*]

M. Utkan Gezer[†] A. C. Cem Say

Department of Computer Engineering, Boğaziçi University, İstanbul, Türkiye

revisions 1st July 2024, 25th Feb. 2025; accepted 2nd Oct. 2025.

Interactive proof systems whose verifiers are constant-space machines have interesting features that do not have counterparts in the better studied case where the verifiers operate under reasonably large space bounds. The language verification power of finite-state verifiers is known to be sensitive to the difference between private and public randomization. These machines also lack the capability of imposing worst-case superlinear bounds on their own runtime, and long interactions with untrustable provers can involve the risk of being fooled to loop forever. We analyze such verifiers under different bounds on the numbers of private and public random bits that they are allowed to use. This separate accounting for the private and public coin budgets as resource functions of the input length provides interesting characterizations of the collections of the associated languages. When the randomness bound is constant, the verifiable class is NL for private-coin machines, but equals just the regular languages when one uses public coins. Increasing the public coin budget while keeping the number of private coins constant augments the power: We show that the set of languages that are verifiable by such machines in expected polynomial time (with an arbitrarily small positive probability of looping) equals the complexity class P. This hints that allowing a minuscule probability of looping may add significant power to polynomial-time finite-state automata, since it is still not known whether those machines can verify all of P when required to halt with probability 1, even with no bound on their private coin usage. We also show that logarithmic-space machines which hide a constant number of their coins are limited to verifying the languages in P.

Keywords: interactive proof systems, probabilistic finite automata, multihead finite automata, alternating finite automata

1 Introduction

In addition to providing a new perspective on the age-old concept of proof, and offering possibilities for weak clients to check the correctness of difficult computations that they delegate to powerful servers, interactive proof systems also play an important role in the characterization of computational complexity classes [16, 17, 22]. These systems involve a computationally weak verifier (a probabilistic Turing machine with small resource bounds) engaging in a dialogue with a strong but possibly malicious prover,

[*]This research was partially supported by Boğaziçi University Research Fund Grant Number 19441. An earlier version of this paper [12] was presented in the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13–15, 2023. This is a substantially extended version.

[†]Utkan Gezer's participation in this work is supported by the Turkish Directorate of Strategy and Budget under the TAM Project number 2007K12-873.

ISSN 1365–8050

whose aim is to convince the verifier that a common input string is a member of the language under consideration. If the input is a non-member, the prover may well lie during this exchange to mislead the verifier to acceptance, or to trick it into running forever instead of rejecting. Interestingly, this setup allows the weak machines to be able to verify (that is, to determine the membership status of any given string with low probability of being fooled) a larger class of languages than they can manage to handle in a stand-alone fashion, *i.e.*, without engaging with a prover.

Several specializations of the basic model described above have been studied until now. One parameter is whether the prover can directly see the outcomes of the random choices made by the verifier or not. A *private-coin* system hides the results of the verifier's coin flips from the prover, and the verifier only sends information that it deems necessary through the communication channel. *Public-coin* systems, on the other hand, hide nothing from the prover, who can be assumed to observe the coin flips and deduce the resulting changes to the configuration of the verifier as they unfold. It is known [9] that private-coin systems are more powerful (*i.e.*, can verify more languages) than public-coin ones when the verifiers are restricted to be constant-space machines, but this distinction vanishes when the space restriction is lifted [15].

In this paper, we study the capabilities of constant-space verifiers (essentially, two-way probabilistic finite-state automata) which are allowed to hide some, but not necessarily all, of their coin flips from the prover. The separate accounting for the private and public coin budgets as resource functions of the input length provides new characterizations of the collections of associated languages in terms of well known complexity classes.

The rest of the paper is structured as follows: Section 2 provides the preliminaries. A definition of interactive proof systems which allows separate accounting of private and public coin usage is given in Section 2.1, together with notation that generalizes the standard IP classes in an accordingly parameterized manner. Since some of our constructions involve verifiers which do not halt with probability 1, we introduce notation that enables us to talk about resource bounds that apply in the (highly probable) cases where the machines do halt. Our main result uses a technique that depends on the relationship between logarithmic-space computation and multihead finite automata. Section 2.2 contains a quick review of that fact. Section 2.3 describes two methods for implementing probabilistic "alarm clocks" for later use in our proofs. Section 2.4 presents the definition of the two-way alternating finite automaton model, which will be employed in the proof of Theorem 6.

We start Section 3 by noting that finite-state verifiers which are only allowed to flip $O(1)$ private coins are already known to outperform deterministic verifiers, and consider the analogous question about public coins. It turns out (Theorem 6) that finite-state verifiers tossing only $O(1)$ public coins can verify only regular languages. This section also includes a technical result (Theorem 8) establishing that one has to consider bounds on expected, rather than worst-case, runtimes of finite-state verifiers to be able to conduct a meaningful study of superlinear resource bounds.

Our main result (Theorem 9, Section 4) is a demonstration of the equality of the set of languages verifiable by polynomial-time finite-state machines that flip $O(1)$ private coins and polynomially many public coins to the complexity class P. (The verifiers in question are allowed to be fooled by malicious provers to run forever without halting, but the probability of this error can be bounded below any desired positive value. The bounds on the runtime and number of public coin flips apply to halting computations.) The proof builds on previous work [3, 11, 17, 21] on logarithmic-space verification, prover-aided simulation of multihead automata by single-head probabilistic finite automata, and specialized loop avoidance techniques for such verifiers. Notably, Theorem 9 hints that allowing a minuscule probability of looping may

add significant power to polynomial-time finite-state machines, since it is still not known whether those machines can verify all of P when required to halt with probability 1, even with no bound on their private coin usage. We also show that logarithmic-space machines which hide a constant number of their coins are limited to verifying the languages in P, even with arbitrarily large time and public-coin budgets.

Section 5 lists some open problems. Detailed constructions involved in some of the proofs are presented in the Appendix to avoid cluttering the main text.

2 Background

2.1 Interactive proof systems

We start by providing definitions of interactive proof systems and related language classes that are general enough to cover finite-state verifiers with both private and public coins, as well as the more widely studied versions with greater memory bounds [2, 9, 16].

An *interactive proof system (IPS)* for some language L is a protocol between a *verifier* and a *prover*. The verifier is a probabilistic Turing machine tasked with determining whether the prover's argument is sufficiently convincing to conclude that the input string is a member of L. The messages of the prover at any step during its communication with the verifier are determined by a (not necessarily computable) function of the input string and the transcript of the communication up to that point. This function maximizes the probability that the verifier accepts if the input is in L, and minimizes the probability that it rejects otherwise.

The verifier has a read-only input tape, a read-write work tape, and a read-write communication cell for interacting with the prover. It is modeled as a 6-tuple $(Q, \Sigma, \Phi, \Gamma, \delta, q_0)$, with the following components:

1. Q is the finite set of states. The following subsets of Q are not necessarily disjoint, unless specified otherwise:
   - Q_{pri} is the set of states that flip private coins.
   - Q_{pub} is the set of states that flip public coins.
   - Q_{com} is the set of communication states, *i.e.*, those that write to the communication cell. $Q_{\mathrm{pub}} \subseteq Q_{\mathrm{com}}$.
   - $\{ q_{\mathrm{acc}}, q_{\mathrm{rej}} \}$ are the accept and reject states, respectively. $q_{\mathrm{acc}}, q_{\mathrm{rej}} \notin Q_{\mathrm{pri}} \cup Q_{\mathrm{com}}$.
2. Σ is the input alphabet.
3. Φ is the work tape alphabet, containing the special blank symbol $\sqcup$.
4. Γ is the communication alphabet. $\sqcup \in \Gamma$.
5. δ is the transition function, described below.
6. q_0 is the initial state. $q_0 \in Q$.

The computation of a verifier on an input string $w \in \Sigma^*$ is initialized as follows:

- The input tape contains $\triangleright w \triangleleft$, where $\triangleright, \triangleleft \notin \Sigma$ are the left and right end-markers, respectively. The input tape head starts on the left end-marker.
- The work tape is filled with blank symbols, and the work tape head is positioned at the beginning of the tape.

- The communication cell is also blank.

Let $\Sigma_{\bowtie} = \Sigma \cup \{ \rhd, \lhd \}$. Let $\Delta = \{ -1, 0, +1 \}$ be the set of possible head movements, where -1 means "move left", 0 means "stay put", and $+1$ means "move right". Let $\overline{A}$ denote the complement of a set A, *e.g.*, $\overline{Q_{\text{pri}}} = Q \setminus Q_{\text{pri}}$. The computation of the verifier is governed by its transition function δ, which is defined in parts in the manner depicted in Table 1 and explained below.

Tab. 1: Parts of the verifier's transition function.

Case	**Mapping**
$q \in Q_{\text{pri}} \cap Q_{\text{pub}}$	$\delta(q, \sigma, \phi, \gamma, b_{\text{pri}}, b_{\text{pub}}) = (q', \phi', \gamma', d_i, d_w)$
$q \in Q_{\text{pri}} \cap \overline{Q_{\text{pub}}} \cap Q_{\text{com}}$	$\delta(q, \sigma, \phi, \gamma, b_{\text{pri}}) = (q', \phi', \gamma', d_i, d_w)$
$q \in Q_{\text{pri}} \cap \overline{Q_{\text{pub}}} \cap \overline{Q_{\text{com}}}$	$\delta(q, \sigma, \phi, \gamma, b_{\text{pri}}) = (q', \phi', d_i, d_w)$
$q \in \overline{Q_{\text{pri}}} \cap Q_{\text{pub}}$	$\delta(q, \sigma, \phi, \gamma, b_{\text{pub}}) = (q', \phi', \gamma', d_i, d_w)$
$q \in \overline{Q_{\text{pri}}} \cap \overline{Q_{\text{pub}}} \cap Q_{\text{com}}$	$\delta(q, \sigma, \phi, \gamma) = (q', \phi', \gamma', d_i, d_w)$
$q \in \overline{Q_{\text{pri}}} \cap \overline{Q_{\text{pub}}} \cap \overline{Q_{\text{com}}} \setminus \{ q_{\text{acc}}, q_{\text{rej}} \}$	$\delta(q, \sigma, \phi, \gamma) = (q', \phi', d_i, d_w)$

At each step of its computation, the verifier does the following:

- It reads the symbols $(\sigma, \phi, \gamma) \in \Sigma \times \Phi \times \Gamma$ from its input tape, work tape, and communication cell respectively. If its current state q is in Q_{pri}, it tosses a private coin, obtaining the outcome $b_{\text{pri}} \in \{ 0, 1 \}$. If q is in Q_{pub}, it tosses a public coin, obtaining the outcome $b_{\text{pub}} \in \{ 0, 1 \}$.
- The value of δ corresponding to this tuple (see the template of the corresponding part of δ in Table 1) dictates that the machine will switch to state $q' \in Q$, write $\phi' \in \Phi$ to its work tape, and move its input and work tape heads in the directions $d_i, d_w \in \Delta$, respectively. The verifier overwrites the communication cell with $\gamma' \in \Gamma$ if $q \in Q_{\text{com}}$. If $q \in Q_{\text{pub}}$, the outcome of the public coin flip (b_{pub}) is also communicated to the prover automatically through a separate channel.

Each time the verifier writes a symbol to the shared communication cell, the prover overwrites that symbol with a (possibly different) member of Γ, determined as a function of w, the history of public coin outcomes, and the communication symbols written to the communication cell up to that point. Since the prover is supposed to embody the optimal function to convince (or, when the input string is not in the language L, to deceive) the verifier, one can assume that the prover knows the algorithm of the verifier. Note, however, that the prover does not see the private coin outcomes, tape head positions, work tape content, and internal state of the verifier.

We define the *configuration* of a verifier at any given time as the tuple composed of its state, the contents of its work tape, the symbol in the communication cell, and the positions of its input and work tape heads.

A verifier halts with acceptance (rejection) when it executes a transition entering q_{acc} (q_{rej}). Any transition that moves the input head beyond an end-marker delimiting the string written on the read-only input tape leads to a rejection, unless that last move enters q_{acc}. Any transition that attempts to move the

work head off the left end of its tape also leads to rejection. Note that the verifier may possibly never halt, in which case it is said to be looping.

We say a verifier V in an IPS *verifies a language L with error $\varepsilon = \max(\varepsilon^+, \varepsilon^-)$* if there exist numbers $\varepsilon^+, \varepsilon^- < 1/2$ satisfying the following:

- There exists a prover P such that, for all input strings $w \in L$, V halts by accepting with probability at least $1 - \varepsilon^+$ when started on w and interacting with P.
- For all provers P^* and for all input strings $w \notin L$, V halts by rejecting with probability at least $1 - \varepsilon^-$ when started on w and interacting with P^*.

The terms ε^+ and ε^- bound the two possible types of error corresponding to failing to accept and reject, respectively.

We will be using the notation $\mathrm{IP}(\mathrm{resource}_1, \mathrm{resource}_2, \ldots, \mathrm{resource}_k)$ to denote the class of languages that can be verified with error ε (for some $\varepsilon < 1/2$) by machines that operate within the resource bounds indicated in the parentheses. These may represent budgets for runtime, work tape (space) usage, and number of public and private random bits, given as a function of the length of the input string, in asymptotic terms. We reserve the symbol n to denote the length of the input string. The terms con, $\log$, linear, and poly will be used to represent the well-known types of functions to be considered as resource bounds, with "con" standing for constant functions of the input length, the others being self evident, to form arguments like "$\mathrm{poly\text{-}private\text{-}coins}$" or "$\mathrm{con\text{-}space}$".[(i)] The symbol ∞ will be used to indicate that no upper bound limits the usage of a resource under consideration, as in "$\infty\text{-}\mathrm{time}$". The absence of a specification for a given type of resource (*e.g.*, private coins) shall indicate that that type of resource is simply unavailable to the verifiers of that class.

By default, a given resource budget should be understood as a worst-case bound, indicating that it is impossible for the verifier to exceed those bounds. To indicate bounds in terms of statistical expectations, we will add the "$_\%$" denotation as a subscript, such as "$\mathrm{poly}_\%\text{-}\mathrm{time}$" to indicate polynomial expected runtime. Some of the interactive protocols to be discussed have the property that the verifier has some small probability of being fooled to run forever by a malicious prover trying to prevent it from rejecting the input. The designer of such protocols can set a parameter $\varepsilon^{\mathrm{loop}}$, representing an upper bound to this probability, to any desired small positive value. The denotation "$^\%$" will be used, this time as a superscript, to mark that the indicated amount corresponds to such a machine's expected consumption of a specific resource with the remaining (high) probability that is at least $1 - \varepsilon^{\mathrm{loop}}$ for any input string. For instance, "$\mathrm{poly}^\%\text{-}\mathrm{time}$" will indicate that the verifier's expected runtime is polynomially bounded with probability almost, but possibly not exactly, 1.

Some verifier algorithms we describe in this article will have "dials" that allow one to tune them to set their error bounds to any desired (positive) constant. Such verifiers will be qualified as verifying their language with *(arbitrarily) low error*. To denote classes of languages that are verifiable with arbitrarily low error, we will be using the $\mathrm{IP}_*(\ldots)$ denotation instead of $\mathrm{IP}(\ldots)$.

We will also be considering verifiers with one-sided error: A verifier V for language L is said to have *perfect completeness* if there exists a prover whose interaction with V leads it to accept any string in L with probability 1, and no prover can convince L to accept a string that is not in L with probability 1.

(i) Constant-space verifiers which do not use their work tape at all are also called *finite-state verifiers*. (A Turing machine that is restricted to scan $O(1)$ work tape cells can be simulated by a finite-state machine.) Stand-alone finite-state verifiers (that do not "listen" to what the prover says) are called *probabilistic finite automata*.

In one of our proofs in Section 4, we will be considering finite-state verifiers with multiple input tape heads that the machine can move independently of one another. This type of verifier can be modeled easily by modifying the tuples in the transition function definitions above to accommodate more scanned input symbols and input head directions. Section 2.2 provides more information on finite automata with multiple input heads and their relationships with the standard Turing machine model.

2.2 Multihead finite automata and log-space machines

A *k-head nondeterministic finite automaton* ($2\mathrm{nfa}(k)$) is a nondeterministic finite-state machine with k read-only heads that move on an input string flanked by two end-marker symbols. Each head can be made to stay put or move to an adjacent tape cell in each computational step. Formally, a $2\mathrm{nfa}(k)$ is a 4-tuple (Q, Σ, δ, q_0), where

1. Q is the finite set of internal states, which includes the two halting states q_{acc} and q_{rej},
2. Σ is the finite input alphabet,
3. $\delta\colon Q \times \Sigma_{\rhd\lhd}^k \to \mathcal{P}\big(Q \times \Delta^k\big)$ is the transition function describing the sets of alternative moves the machine may perform at each execution step, where each move is associated with a state to enter and a movement direction for each head, given the machine's current state and the list of symbols that are currently being scanned by the k input heads, and $\Sigma_{\rhd\lhd}$ and Δ are as defined previously in Section 2.1, and
4. $q_0 \in Q$ is the initial state.

Given an input string $w \in \Sigma^*$, a $2\mathrm{nfa}(k)$ $M = (Q, \Sigma, \delta, q_0)$ begins execution from the state q_0, with $\rhd w \lhd$ written on its tape, and all k of its heads on the left end-marker. At each step, M nondeterministically updates its state and head positions according to the choices dictated by its transition function. Computation halts if one of the states q_{acc} or q_{rej} has been reached, or a head has moved beyond either end-marker.

M is said to *accept* w if there exists a sequence of nondeterministic choices where it reaches the state q_{acc}, given w as the input. M is said to *reject* w if every sequence of choices either reaches q_{rej}, ends with a transition whose associated set of choices is $\emptyset$, or by a head moving beyond an end-marker without a halting state being entered. M might also loop on the input w, neither accepting nor rejecting it.

The *language recognized by* M is the set of strings that it accepts.

Let $\mathcal{L}(2\mathrm{nfa}(*))$ denote the set of languages that have a $2\mathrm{nfa}(k)$ recognizer (for some $k > 0$), and $\mathcal{L}(2\mathrm{nfa}(*), \text{linear-time})$ denote the set of languages that have a $2\mathrm{nfa}(k)$ recognizer running in linear time, regardless of the nondeterministic choices it makes.

Our characterization of P (Section 4) makes use of the equivalence of multiple input heads and logarithmic amounts of memory, discovered by Hartmanis [18]. The following theorem, whose proof is given in Appendix A, reiterates one direction of that result in detail, and will be useful for our purposes.

Theorem 1. *Any language recognized by a Turing machine that uses at most $\lfloor \log(n+2) \rfloor$ space with a work tape alphabet of size at most 2^c (for some integer constant $c > 0$) and within $t(n)$ time can also be recognized by a $(c+5)$-head finite automaton within $t(n) \cdot ((2c + 3/2)n + 2c + 2)$ time.*

2.3 Probabilistic clocks

A logarithmic-space Turing machine can time its own execution to satisfy any desired polynomial bound by counting up to that bound in the logarithmic space available. The constant-space machines we construct

in Section 4 will employ a different technique using randomness, which is illustrated in the following lemma, to obtain the same bound on expected runtime.

Lemma 2. *For any integer $t > 0$, integer-valued function $f(n) \in O(n^t)$, and desired "error" bound $\varepsilon_{\text{premature}} > 0$, there exists a probabilistic finite automaton with expected runtime in $O\big(n^{t+1}\big)$, such that the probability that this machine halts in fewer than $f(n)$ time-steps is $\varepsilon_{\text{premature}}$.*

The proof of Lemma 2, based on the idea [1, 9] of the machine performing a sequence of random walks with its head on the input tape, is given in Appendix B.

In the proof of Lemma 12 in Section 4, we describe a probabilistic log-space machine that uses a timer which is expected to run out after at least $2^{2^{p(n)}}$ "ticks", where p is some polynomial, and n is the length of the machine's input string. We describe the main idea, which was inspired by a similar construction in [4], below; a detailed analysis can be found in Appendix C.

It will be helpful to imagine that we have fair coins of two colors, say, red and blue. The timer mechanism consists of a subroutine that embodies a single tick, and a global Boolean variable that tracks whether all the red coins flipped since the last reset of this variable came out heads. Each call of this subroutine, which is named TICK, flips one red coin and $p(n)$ blue coins. If one or more blue coins come out tails, the subroutine call ends with the timer still running. If all the blue coin outcomes are heads, but the Boolean variable indicates that one or more red coins flipped since the last reset came out tails, the variable is reset, and the subroutine ends with the timer still running. If all the blue coin outcomes are heads, *and* if all the red coins flipped since the last reset also came out heads, the timer runs out.

The expected number of ticks is at least double exponential in $p(n)$, since the timer runs out when an all-heads sequence of red coins is observed, and the expected length of this critical sequence is exponential in $p(n)$.

2.4 Alternating finite automata

A *two-way alternating finite automaton* (2afa) is a 4-tuple (Q, Σ, δ, q_0), where

1. Q, the finite set of states, is the union of the following disjoint subsets:
   - $Q_\exists$ is the set existential states,
   - $Q_\forall$ is the set of universal states,
   - $\{\, q_{\text{acc}}, q_{\text{rej}} \,\}$ are the accept and reject states, respectively,
2. Σ is the finite input alphabet,
3. $\delta\colon Q \times \Sigma_{\rhd\lhd} \to \mathcal{P}(Q \times \Delta)$, which is identical in format to the corresponding item in the definition of a $2\text{nfa}(1)$ (Section 2.2), is the transition function describing the sets of alternative moves that the machine may perform at each execution step, and
4. $q_0 \in Q$ is the initial state.

As a computational model, the 2afa is a generalization the $2\text{nfa}(1)$. A $2\text{nfa}(1)$ is simply a 2afa whose set of universal states is empty, as will be evident from the definition (adapted from [8]) of string acceptance by 2afa's to be presented below.

The *configuration* of a 2afa $M = (Q, \Sigma, \delta, q_0)$ at any given time is the pair composed of M's state and its input tape head position. In its initial configuration, M is in state q_0, with the head positioned on the

left end-marker. A configuration (q, i) is said to be *accepting* (resp. *rejecting*) if q is $q_{\rm acc}$ (resp. $q_{\rm rej}$). A configuration is said to be *universal* (resp. *existential*) if its state is universal (resp. existential). Given an input string w, a configuration β is said to be a *successor* of a configuration α of M, denoted $\alpha \vdash_{M,w} \beta$, if β can follow α immediately according to δ. Note that a configuration may have multiple successors, or none at all.

M is defined to *accept* w if and only if $l_{M,w}((q_0, 0))$ equals `true`, where the function $l_{M,w}$ is defined recursively as follows:

$$l_{M,w}(\alpha) = \begin{cases} \texttt{true}, & \text{if } \alpha \text{ is accepting,} \\ \texttt{false}, & \text{if } \alpha \text{ is rejecting,} \\ \bigvee\limits_{\alpha \vdash_{M,w} \beta} l_{M,w}(\beta), & \text{if } \alpha \text{ is existential,} \\ \bigwedge\limits_{\alpha \vdash_{M,w} \beta} l_{M,w}(\beta), & \text{if } \alpha \text{ is universal.} \end{cases}$$

The *language recognized by* M is the set of input strings that it accepts.

A good way to understand the relationship of a 2afa with the strings it accepts is to visualize a "prover" (whose aim is to make the machine accept the input) as dictating which move to make when the transition function allows multiple outgoing choices from a configuration with an existential state, and a "refuter" (with the opposite aim) as similarly dictating moves from configurations with universal states. The input string w is accepted by the machine if and only if the prover has a winning strategy, whereby it can lead the machine from the initial configuration associated with w to an accepting configuration, no matter what moves are made by the refuter, in this perfect information game.

It is known [20] that the class of all languages recognized by 2afa's equals REG, *i.e.*, the class of regular languages.

3 Private vs. public coins and worst-case time bounds

In their seminal paper, Dwork and Stockmeyer [9] showed that finite-state verifiers employing only private coins are strictly more powerful than those using only public coins, even when the former machines are bounded to operate in polynomial expected time:

Fact 3.

$$\mathrm{IP}(\text{con-space}, \infty\text{-public-coins}, \infty\text{-time}) \subsetneq \mathrm{IP}_*(\text{con-space}, \mathrm{poly}_{\%}\text{-private-coins}, \mathrm{poly}_{\%}\text{-time}).$$

A witness for the inequality in Fact 3 is the language of palindromes,

$$L_{\rm pal} = \left\{ \, w \;\middle|\; w \in \{\, 0, 1 \,\}^*, w = w^R \, \right\},$$

where x^R denotes the reverse of string x. The constant-space verifier provided for $L_{\rm pal}$ by Dwork and Stockmeyer uses only a constant number of private coins, irrespective of the length of the input. This inspired the study [11, 13, 21] of machines that flip a fixed number of coins, leading to the following results:

Fact 4. $\mathcal{L}(\mathrm{2nfa}(*), \text{linear-time}) \subseteq \mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, \text{linear}^{\%}\text{-time})$ [11].[(ii)]

[(ii)] Recall from the definition of our IP complexity class notation in Section 2 that the verifier's runtime can be infinite with at most a small probability $\varepsilon^{\text{loop}}$, and its expected runtime is bounded as indicated with the remaining large probability.

Fact 5. $\mathrm{IP}(\text{con-space}, \text{con-private-coins}, \infty\text{-time}) = \mathrm{NL}$ [21].

In contrast to Fact 5, we now show that the ability to use constant number of *public* coin flips provides no additional language verification power over determinism to constant-space machines.

Theorem 6. $\mathrm{IP}(\text{con-space}, \text{con-public-coins}, \infty\text{-time}) = \mathrm{REG}$.

Proof: Let V be a finite-state machine that verifies a language L with some error bound $\varepsilon < 1/2$, flipping at most $r > 0$ public coins for any input. Let the prover that interacts with V be named P.

We will construct a two-way alternating finite automaton M that recognizes L. This will be enough to conclude that L is regular, by the fact [20] that two-way alternating finite automata recognize all and only the regular languages.

The construction is based on the following idea: Consider the set of all possible bit sequences of length r. Each execution of V would use (a prefix of) one of these sequences as its public coin outcomes during its interaction with the prover. Any input string w is in L if and only if a majority of the members of this set of sequences leads V to acceptance during this interaction. M will be designed so that it accepts w if and only if it verifies that this condition is satisfied. Note that simply simulating V's behavior sequentially, on one coin sequence after another, would not work, since this would give the simulated prover the opportunity to cheat by violating the condition that its responses to V given two coin sequences with the same prefix (like $p0$ and $p1$, for some prefix $p \in \{0,1\}^*$) should be identical up to the point when the coin sequences finally differ. M will use the "parallel computation" nature of alternation to enforce this consistency on the simulated prover.

M starts with an existential state, where it guesses a prefix-free set S of public coin sequences (of length at most r) whose probabilities add up to a value greater than $1/2$. (Since there are only finitely many sets of bit sequences with this property, this choice can be made in a single step.) Recall that such choices of moves from existential states can be viewed as being made by a prover, which we will name P_M. By this first choice, P_M would be claiming that all of these coin sequences would lead V to acceptance in an interaction with P.

M attempts to verify this claim: M's transition function is based on that of V; essentially, M uses alternation to create a parallel simulation of V (and its interaction with P) on all the coin sequences in the set S obtained at the start. At every step, M keeps a record of the (simulated) coin sequence used up to that point in the present branch of its computation. Whenever its simulation of V reaches a new coin-flipping state, M checks this record and *rejects* if it determines that it is in a branch that would use a coin sequence that is not a member of S. If only one of the outcomes of the coin to be flipped at the present state is consistent with S, M plugs that bit into V to advance the simulation. If, on the other hand, *both* of the 0 and 1 outcomes are consistent with S, M makes a universal choice (controlled by the refuter R_M) to determine which bit to feed V. Each communication symbol sent by P to V is obtained by an existential choice (as if P_M is supplying that information). The accept state of M corresponds to the accept state of V.

We see that M recognizes L by noting that any w is in L if and only if there exists a set of random bit sequences with total probability greater than $1/2$ such that all members of this set lead V to acceptance of input w after an interaction with P, and M accepts all and only such input strings by design.

For completeness, a formal description of the $2\mathrm{afa}$ M is presented in Appendix D. □

We will focus on the computational power of constant-space verifiers with a fixed private-coin budget when they are also allowed to use public coins. Although some of Dwork and Stockmeyer's results

(*e.g.*, [9, Theorem 3.12]) involve such "mixed-coin" constructions, the asymptotic bounds on private and public coin tosses are equal in their machines. In those setups, all public coins can simply be replaced by private coins by a straightforward modification that increases the runtime of the protocol by a constant factor. The proof of the following lemma is in Appendix E.

Lemma 7. *The following is true for all functions s, f, g, and t:*

$$\mathrm{IP}_*(O(s(n))\text{-space}, O(f(n))\text{-private-coins}, O(g(n))\text{-public-coins}, O(t(n))\text{-time}) \subseteq \mathrm{IP}_*(O(s(n))\text{-space}, O(f(n)+g(n))\text{-private-coins}, O(t(n))\text{-time}).$$

This inclusion also holds for the $\mathrm{IP}(\ldots)$ *variants of the classes, and when worst-case time bounds are replaced with those corresponding to expected usage.*

Lemma 7 implies that allowing the finite-state verifiers in the constructions of Facts 4 and 5 to use a constant number of public coins in addition to their private-coin budgets would not enlarge the corresponding classes of verified languages.

Our main result to be presented in the next section involves constant-space verifiers that flip a constant number of private coins and superlinear amounts of public coins. We note that expected (rather than worst-case) time bounds are appropriate for studying that scenario, since the imposition of any worst-case bound on the runtime of a finite-state verifier precludes it from benefiting from superlinear amounts of any kind of resource.

Theorem 8. *For any time complexity function t,*

$$\mathrm{IP}_*(\text{con-space}, \infty\text{-private-coins}, \infty\text{-public-coins}, O(t(n))\text{-time}) \subseteq \mathrm{IP}_*(\text{con-space}, \text{linear-private-coins}, \text{linear-public-coins}, \text{linear-time}).$$

The same is true also for the $\mathrm{IP}(\ldots)$ *variants of the two classes.*

Proof: The number of configurations available to a constant-space verifier V running on an input of length n is in $O(n)$.

If V's transition function allows it to enter the same configuration more than once with nonzero probability for some input string, it will do so arbitrarily many times with positive probability, exceeding any preset time bound. Therefore, all constant-space verifiers with a worst-case time bound $t(n)$ are actually limited to run within a linear bound. The linear bounds on coin usage are established by noting that a machine that can run for at most T steps is restricted to flipping at most T coins of any type. □

4 A new characterization of P

Let us now examine finite-state verifiers employing a constant amount of private coins and an expected polynomial amount of public coins (unless they are tricked to loop forever with some arbitrarily small probability). This setup turns out to provide a new characterization of the complexity class P, corresponding to the collection of languages decidable by deterministic Turing machines in polynomial time and space.

Theorem 9. *The following three classes are equal:*

1. $\mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, \text{poly}^{\%}\text{-public-coins}, \text{poly}^{\%}\text{-time})$
2. $\mathrm{IP}(\text{log-space}, \text{con-private-coins}, \infty\text{-public-coins}, \infty\text{-time})$
3. P

Theorem 9 follows from Lemmas 10 and 12.

Lemma 10. $\mathrm{P} \subseteq \mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, \text{poly}^{\%}\text{-public-coins}, \text{poly}^{\%}\text{-time})$.

Proof: Let L be any language in P. As Goldwasser et al. have proven in [17], there exists a public-coin verifier V_1 verifying L with the following properties:

- V_1 has perfect completeness;
- For any string w not in L and any prover P^*, the probability that V_1 is convinced to accept w by P^* is at most $1/2$; and
- There exists an integer $t > 1$ such that V_1 uses $O(\log n)$ space and $O(n^t)$ time for any input of length n. (These are worst-case bounds.)

We will assume that exactly $\lfloor \log(n+2) \rfloor$ cells are used in the work tape of V_1, and a multi-track alphabet (*e.g.*, as in [18]) is used to accommodate for the required amount of memory.

In the following discussion, let any prover facing V_1 be called P_1.

There exists a constant-space, public-coin, k-head verifier V_2 that can verify L by simply executing V_1's program, simulating V_1's logarithmic-length work tape by the technique of Theorem 1. (Note that k depends on the precise worst-case memory requirement of V_1.) Since the simulation is direct and does not involve any additional use of randomness, V_2 verifies L with the same amount of one-sided error as V_1. The only time overhead is caused by the simulation of the logarithmically bounded memory, so, by Theorem 1, V_2 will complete its execution in $O\left(n^{t+1}\right)$ time. V_2's prover, say, P_2, is supposed to follow the same protocol as P_1.

We now describe V_3, a constant-space, single-head verifier that uses a constant number of private coins, in addition to the public coins that it flips at almost *every* step, to emulate V_2's verification of L.[iii]

V_3 (Algorithm 1) performs the following m-round procedure:[iv]

Each round begins with V_3 flipping r of its private coins. Using this randomness, it picks one of the k heads of V_2. Each head has the same very small probability $p = 2^{-r}$ of being selected. (How V_3 operates with the remaining high probability $1 - kp$ will be explained later.) V_3 then engages in an interaction with its own prover, say, P_3, to simulate the execution of V_2, including V_2's interaction with P_2 about the input string.

Note that V_2 flips coins in some, but not necessarily all of its computation steps, whereas V_3 flips a public coin at each step of the simulation. Only some of these coin outcomes are used to stand in for V_2's public coins. What V_3 does with the remaining public random bits will be explained below.

To simulate V_2, V_3 traces the selected head of V_2 with its own single head, and relies on P_3 to provide an unbroken stream of information about what the other heads of V_2 would be reading at every step of its execution. In its response to any coin flip of V_3, P_3 is expected to transmit its claims about the readings

[iii] This is an adaptation of a technique introduced by Say and Yakaryılmaz [21] for simulating a multihead nondeterministic automaton in an interactive proof system whose verifier is a (single-head) probabilistic finite automaton.

[iv] The precise settings of V_3's parameters (like m) will be discussed below.

of all k heads of V_2 at that step of the simulated interaction (and P_2's response to V_2, if the currently simulated transition of V_2 emanated from a communication state). When the simulation arrives at an actual coin-flipping state of V_2, P_3 is supposed to interpret the latest coin outcome as a public coin flipped by V_2.

At any step, V_3 checks the part of P_3's claims regarding the head it had chosen in private, and *rejects* if it sees any discrepancy. If the information sent by P_3 has led the simulation of V_2 to reach acceptance at the current step, V_3 has not been able to catch a lie up to that point, and if this was not the mth round, V_3 moves its head back to the left end of the input tape without flipping coins, and proceeds to the next round.

The probability p_{head} that V_3 will attempt to use its head to check the claims of P_3 in the manner described above is just kp. With the remaining high probability p_{timer} (*i.e.*, $1 - p_{\text{head}}$), V_3 simulates V_2 by relying on P_3's claims about the head readings and P_2 responses blindly, while using its own head and the public coins it flips to simultaneously simulate a probabilistic finite automaton M_{timer} that functions as a timer (Appendix B) in this round. This timer has an expected runtime of $O\left(n^{t+2}\right)$, and exceeds V_2's runtime with probability $1 - \varepsilon_{\text{premature}}$, for some positive $\varepsilon_{\text{premature}}$ that can be set to be arbitrarily close to 0, by the premise of Lemma 2. If the simulation of V_2 fed by P_3 reaches acceptance before the timer

In the following, V_2 is assumed to never utilize its work tape and to always send a dummy symbol (*e.g.*, $\sqcup$) to its prover when it needs it to update its communication cell, without loss of generality. $\{ H_1, H_2, \ldots, H_k, \mathit{TIMER} \}$ is the set of different modes in which V_3 can operate.

Repeat the following for m rounds:
- Use r private coin flips to select $mode$ to be an element of $\{ H_1, \ldots, H_k \}$ with $p = 2^{-r}$ probability for each, or TIMER with the remaining probability.
- Move the input head to the left end-marker.
- Initialize a simulation of V_2, using a field γ_2 to keep its communication cell content.
- Communicate with P_3 for it to reply with its claims about V_2's initial head readings.
- If $mode = \mathit{TIMER}$, initialize a parallel simulation for M_{timer}.
- Repeatedly execute the following move, which advances the simulation(s) by one step, until V_2 accepts:
  - (All actions described for this iteration are executed in a single verifier transition.)
  - If $mode = \mathit{TIMER}$ and M_{timer} has halted, *reject*.
  - If V_2 has rejected, *reject*.
  - Let γ_3 denote the symbol in the communication cell.
  - Extract P_3's claims about V_2's head readings from γ_3.
  - If $mode = H_i$, and the scanned input symbol does not match P_3's claim, *reject*.
  - If V_2 communicated in the previously simulated step, extract the new value for γ_2 from γ_3, retaining its old value otherwise.
  - Flip a public coin.
  - Advance the simulation(s) utilizing the extracted information about V_2's head readings, communication symbol and the coin outcome, moving the input head to imitate the ith head of V_2 if $mode = H_i$, and to imitate M_{timer}'s head otherwise.

Accept.

Algorithm 1: A constant-space, single-head verifier V_3.

runs out, then V_3 completes this round of verification without rejecting. Otherwise (if the timer runs out before the simulation ends), V_3 *rejects*.

V_3 *accepts* if it does not reject for m rounds of verification. The total number of private coins used is mr.

The rest of the proof is an analysis of the error probability and runtime of V_3.

Arbitrarily small verification error. For any input string that is a member of L, P_3 can simply tell V_3 the truth about what V_2 would read with its k heads, and emit the messages that P_2 would send to V_2 alongside those readings. Even when communicating with such a truthful P_3, V_3 may erroneously reject at any given round, due to a premature timeout of the probabilistic timer. The probability of that is at most $p_{\text{timer}} \cdot \varepsilon_{\text{premature}}$. To accept a member of L, V_3 must go through m consecutive rounds of verification without this event occurring. The maximum probability with which V_3 can fail to accept a string in L is therefore

$$\varepsilon_3^+ \leq 1 - (1 - p_{\text{timer}} \cdot \varepsilon_{\text{premature}})^m.$$

We will now determine a bound for ε_3^-, the probability with which V_3 may fail to reject an input string that is not a member of L. Unlike ε_3^+, ε_3^- is a value that P_3 "wants" to maximize, so one has to be careful to include all possible strategies for the prover in this consideration. The program of V_3 dictates that there are exactly $m + 1$ different ways (corresponding to alternative strategies for P_3) in which V_3 may fail to reject a string w not in L:

- V_3 may pass all m rounds by carrying out an assisted simulation of V_2 that ends in what seems to it to be acceptance. We denote this event by P^m.
- V_3 may be tricked (as will be described below) by P_3 into entering an infinite loop during the $j+1$st round, after passing the first j rounds, for $j \in \{\, 0, \ldots, m-1 \,\}$. Such an event will be denoted by $\mathsf{P}^j\mathsf{L}$.

Recall that P_3 is expected to feed V_3 with information about the symbols that would be scanned by V_2's input heads at any point of its execution, and the transmissions of P_2. For $w \notin L$, if P_3 always tells the truth about the head readings, V_3 can pass any round with the probability (at most 2^{-1}) that V_2 would be deceived about w, and finally accept with probability at most 2^{-m}. Since P_2's messages are optimal for convincing V_2, whose program is hard-coded in V_3, P_3's only option for achieving better odds of deceiving V_3 is by lying about the head readings of V_2.

Since V_3 is able to track only one of V_2's input heads with its own head, it is dependent on P_3 to carry out the simulation. If V_3 has picked some head h of V_2 to track, a lie by P_3 about the symbol seen at that step by some other head h' of V_2 may be enough to deceive V_3 into eventually believing that V_2 accepts w in that round. V_3 can catch such a lie only if it has previously picked the head that is being lied about, and detects the discrepancy between P_3's claim and its own scan.

Unlike P_1 and P_2, which deal with verifiers that have sufficient memory to keep track of their own runtimes, P_3 also has the capability to trick V_3, whose number of different possible configurations is, in general, less than the runtime of V_2, which it is supposed to be simulating, into running forever. It is common (see, for instance, Appendix A) for multihead finite automata to require a head to wait at a tape location while another head is walking towards a specific symbol, *e.g.*, an end-marker. If V_3 has picked to track such a waiting head of V_2 with its own head, it can be tricked into getting stuck in that configuration by an endless stream of claims from P_3 that the other head is still walking in the required direction. V_3

can catch such a lie directly, if it has picked the head that is being lied about; or indirectly, if it has been running in "timer mode".

Our analysis of the error probabilities associated with the $m+1$ alternative strategies for the prover mentioned above requires setting bounds for the two following values:

- ε_{L}, the probability that V_3 is tricked to enter an infinite loop in a particular round.
- ε_{P}, the probability that V_3 is led to complete a round without rejection when $w \notin L$.

V_3 cannot run forever in any round in which it has selected to act as a timer. The probability that V_3 has selected to simulate V_2 with a head that P_3 would lie about is at least p. We conclude that

$$\varepsilon_{\mathsf{L}} \leq p_{\text{head}} - p = (k-1) \cdot p.$$

There are two different ways in which the prover can cause V_3 to complete a round without rejection when $w \notin L$:

1. P_3 can be truthful. With probability at most $1/2$, the public coin flips would lead the resulting faithful simulation to end in V_2's accept state.
2. P_3 can lie about a reading of a head of V_2. In this case, V_3 may be tricked into believing that V_2 accepts w if V_3 has not selected to track that head. The probability of this event is at most $1-p$.

In this scenario, P_3 can maximize its chances of leading V_3 to passing a round by basing its decision about whether to lie or not on the public coin outcomes: Call a public coin sequence *lucky* if it leads V_2 to accepting w when guided by P_2. In such cases, honesty is the best P_3 policy, since it involves zero probability of rejection as a result of being caught lying. For unlucky coin sequences, honesty has no chance of success, and P_3 would have to lie for V_3 to complete the round.

To obtain an easy upper bound for ε_{P}, we grant P_3 certain additional "superpowers" that allow it to always make the correct decision in the situations described above: In particular, we assume that P_3 can always correctly foresee whether the entire public coin sequence of the present round will be a lucky one or not, and that it is able to communicate with V_3 (lying about a single V_2 head or reporting all head readings truthfully) accordingly. Under these extremely favorable circumstances, P_3 can manage to be truthful exactly when the public coin sequence of the present round is a lucky one, and lie in the remaining cases. The probability of a lucky sequence is, as mentioned before, at most $1/2$. We therefore have

$$\varepsilon_{\mathsf{P}} \leq \frac{1}{2} + \left(1 - \frac{1}{2}\right) \cdot (1-p) = 1 - \frac{p}{2}.$$

We can now establish bounds for the probabilities of the $m+1$ "failure to reject" events listed above.

The probability that V_3 is led to accept $w \notin L$ through P^m is

$$\varepsilon_{\mathsf{P}^m} = \varepsilon_{\mathsf{P}}^m \leq \left(1 - \frac{p}{2}\right)^m.$$

The probability that V_3 is made to fail to reject w through $\mathsf{P}^j\mathsf{L}$ ($j \in \{\, 0, \ldots, m-1 \,\}$) is

$$\varepsilon_{\mathsf{P}^j\mathsf{L}} = \varepsilon_{\mathsf{P}}^j \varepsilon_{\mathsf{L}} \leq \left(1 - \frac{p}{2}\right)^j \cdot (k-1) \cdot p.$$

Although the prover has the ability to use the public coins to implement a probabilistic mixture of the $m+1$ mutually exclusive strategies in its repertory, it is easy to see that P_3 must focus only on the strategies associated with the highest "success" probabilities from its own point of view:

$$\varepsilon_3^- \leq \max(\varepsilon_{\mathsf{P}^0\mathsf{L}}, \varepsilon_{\mathsf{P}^1\mathsf{L}}, \ldots, \varepsilon_{\mathsf{P}^m}).$$

Since $\varepsilon_{\mathsf{P}^0\mathsf{L}}$ is clearly the maximum among all $\varepsilon_{\mathsf{P}^j\mathsf{L}}$'s, we have

$$\varepsilon_3^- \leq \max(\varepsilon_{\mathsf{P}^0\mathsf{L}}, \varepsilon_{\mathsf{P}^m}).$$

Let us show that we can tune V_3 so that it verifies L with error $\varepsilon_3 = \max\bigl(\varepsilon_3^+, \varepsilon_3^-\bigr)$, for any small positive value ε_3. Since

$$p = 2^{-r} \quad \text{and} \quad p_{\text{timer}} = 1 - p_{\text{head}} = \frac{2^r - k}{2^r},$$

we have established that the following three quantities should be less than or equal to the allowed error bound:

$$\begin{aligned}
\varepsilon_3^+ &\leq 1 - (1 - p_{\text{timer}} \cdot \varepsilon_{\text{premature}})^m = 1 - \left(1 - \frac{2^r - k}{2^r} \cdot \varepsilon_{\text{premature}}\right)^m \\
\varepsilon_{\mathsf{P}^m} &\leq \left(1 - \frac{p}{2}\right)^m = \left(1 - 2^{-r-1}\right)^m \\
\varepsilon_{\mathsf{P}^0\mathsf{L}} &\leq (k-1) \cdot p = (k-1) \cdot 2^{-r}
\end{aligned}$$

Given some desired ε_3,

1. Set r to a value that ensures $\varepsilon_{\mathsf{P}^0\mathsf{L}}$ does not exceed ε_3.
2. Set m to a value that ensures that $\varepsilon_{\mathsf{P}^m}$ does not exceed ε_3.
3. Set $\varepsilon_{\text{premature}}$ so that ε_3^+ does not exceed ε_3. This can be achieved by arranging the number of states in the implementation of the probabilistic timer, as described in Appendix B.

Note that the parameters r, m, and $\varepsilon_{\text{premature}}$ are independent, allowing each of them to be set without affecting the values of the other two.

Polynomial expected runtime with arbitrarily high probability. With ε_3 (and thereby also the probability of looping) set to a desired tiny value, V_3 will be running for at most m rounds with the remaining high probability. At each of those rounds, V_3 will either complete V_2's simulation in $O\bigl(n^{t+1}\bigr)$ time, or will operate as a probabilistic timer that has expected runtime $O\bigl(n^{t+2}\bigr)$. Thus, it is expected to run in $O\bigl(n^{t+2}\bigr)$ time with probability at least $1 - \varepsilon_3$.

We conclude that $L \in \text{IP}_*(\text{con-space}, \text{con-private-coins}, \text{poly}^{\%}\text{-public-coins}, \text{poly}^{\%}\text{-time})$. □

The following corollary to Lemma 10 depicts that the class NC has finite-state verifiers flipping a fixed number of private coins and with a tighter upper bound on their public coin budget compared to those for P.

Corollary 11. $\mathrm{NC} \subseteq \mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, O\left(n^4\right)^{\%}\text{-public-coins}, O\left(n^4\right)^{\%}\text{-time})$.

Proof: Fortnow and Lund have proven [10] that

$$\mathrm{NC} \subseteq \mathrm{IP}_*(\text{log-space}, O\left(n \log^2 n\right)\text{-public-coins}, O\left(n \log^2 n\right)\text{-time}). \quad (1)$$

The verifiers described in [10] for the right-hand side of (1) have perfect completeness. One can apply the sequence of constructions described in the proof of Lemma 10, starting with such a verifier playing the role played by V_1 in that proof, to obtain the claimed result. □

Lemma 12. $\mathrm{IP}(\text{log-space}, \text{con-private-coins}, \infty\text{-public-coins}, \infty\text{-time}) \subseteq \mathrm{P}$.

Proof: Let V_1 be a log-space verifier that uses at most r private coins and an unlimited budget of public coins to verify a language L with some error bound $\varepsilon_1 < 1/2$, for some constant r. In the following, the prover that V_1 interacts with will be named P_1. The length of the input string is denoted by n.

We will now demonstrate the existence of a log-space verifier V_2 that verifies the same language L with an error bound close to that of V_1 without using any private coins. V_2 also has an unlimited budget of public coins. In a nutshell, V_2 will interact with its prover, which we name P_2, to simulate V_1 parallelly for all the 2^r different possible private random bit sequences. It will feed all of these simulations with the same public random bit sequence that it will generate on the fly, while checking that P_2 is not trying to cheat by supplying messages in a way that would be impossible for P_1 (which, unlike P_2, does not know the current configuration of the verifier it is communicating with), or by attempting to trick V_2 into an unnecessarily long conversation. V_2 will be operating a probabilistic clock to ensure that its expected runtime will be within $2^{2^{\Omega(p(n))}}$, for a suitably chosen polynomial p. When this timer finally runs out, V_2 will pick one of the parallel simulations at random to imitate V_1's decision. We now present a detailed discussion of each of these points.

The interaction between V_2 and P_2. Let $\mathrm{bin}_r(i)$ denote the r-bit binary representation of the natural number $i \in \{ 0, \ldots, 2^r - 1 \}$ (padded with 0's from left as needed). As mentioned above, V_2 will be running 2^r parallel simulations ("sims") of V_1, where the ith simulated verifier S_i is hardwired to use the bits of $\mathrm{bin}_r(i)$ as its private "random" bits. More formally, for each non-halting state q of V_1, each S_i has $r+1$ states of the form (q, π), one for each postfix π of $\mathrm{bin}_r(i)$ to keep track of the remaining unused bits. The initial state of S_i is the pair consisting of the initial state of V_1 and $\mathrm{bin}_r(i)$ itself. For each transition of V_1 that emanates from a state q, consumes a private random bit $b_{\mathrm{pri}} \in \{ 0, 1 \}$, and enters some state q', S_i has corresponding transitions emanating from each state of the form $(q, b_{\mathrm{pri}}\pi)$ (where π is a proper postfix of $\mathrm{bin}_r(i)$), employing no private coin, and entering state (q', π). (Each such S_i transition mimics every other aspect, *e.g.*, communication symbols, input and work tape head actions, etc. of the corresponding V_1 transition faithfully.) Transitions of V_1 that do not consume private random bits are simulated by transitions of S_i that do not change the second component of the machine's state. V_1-transitions entering halting states are mimicked by S_i-transitions entering the unique accept and reject states of S_i. S_i-transitions emanating from (unreachable) states of the form (q, λ), where q is a private-coin-tossing state of V_1, are directed to the reject state.

At each step of its interaction with V_2, P_2 is supposed to supply the responses that P_1 would supply to each sim at the corresponding point, according to the convention that will be explained below. Whenever V_2 is ready to receive the next P_2 symbol, it will use the communication cell to indicate this. As we will see, V_2 will flip public coins in some, but not necessarily all, of its communication steps.

The execution of V_2 (Algorithm 2) proceeds in *stages* delimited by its (public) coin flips. Each stage consists of one or more *segments* delimited by communication steps. Originally starting each S_i from its initial configuration, V_2 proceeds in each segment by going through all the S_i's sequentially,[v] continuing the simulation of each S_i until that sim has accepted, rejected, entered a communication state, or is determined to have entered an infinite loop. (Note that there exists a polynomial f, dependent on V_1, such that a sim that runs for more than $f(n)$ steps without entering a communication state must be in an infinite loop. V_2 has sufficient memory to count deterministically up to $f(n)$, which is simply the number

Initialize simulations of $S_0, \ldots, S_{2^r-1}$ (the sims).
Initialize the *communication partition* (that will keep track of groups of sims with identical communication transcripts) with all sims in the same block.
Initialize the Boolean variable to be used by the TICK subroutine that will implement the timer.
Repeatedly execute the following until the timer runs out: (Each iteration of this loop carries out one *stage* of the protocol, as defined in the text.)
  Initialize the *segment counter* to 0.
  Execute the following until each sim is labeled as halted, looping, or waiting for a coin toss:
    For each S_i that is not labeled as halted, looping, or waiting for a coin toss, do the following:
      Proceed with S_i's simulation until it halts, enters a communication state, or runs for more than the time limit of $f(n)$ steps without entering a communication state. If S_i has halted or exceeded the limit, label it as halted or looping, respectively. If S_i has entered a coin-tossing state, label it as waiting for a coin toss.
    Advance any unlabeled sims (every such sim is in a non-coin-tossing communication state) for one more transition, refine the communication partition by arranging sims transmitting different symbols into different blocks, and prompt P_2 to provide the corresponding responses.
    Use the communication partition to check whether P_2 has attempted to give different responses to sims in the same block. If such a violation is detected, *reject*. Otherwise, update the configurations of the communicating sims with the new responses.
    Increment the segment counter.
    If the value of the segment counter has exceeded $g(n)$, label all as yet unlabeled sims as looping.
  Toss a public coin. Using its outcome, advance all sims that have been waiting for a coin toss for one more transition. Unlabel those sims.
  Refine the communication partition by arranging sims with different communications into different blocks.
  Use the communication partition to check whether P_2 has attempted to give different responses to sims in the same block after the last coin toss. If such a violation is detected, *reject*. Otherwise, update the configurations of the unlabeled sims with the new responses.
  Call the TICK subroutine.
Toss r public coins and accordingly choose an S_i at random. If S_i has rejected or is labeled as looping, *reject*. Otherwise, *accept*.

Algorithm 2: A log-space verifier V_2.

[v] Note that V_2's logarithmic-space budget is sufficient to trace the executions of all the 2^r sims and store the configurations of those that are paused.

of different possible configurations of V_1 on the present input string.) If no sims are in a communication state at the end of this scan of the S_i's, the overall simulation has finished. Otherwise, one or more sims are "waiting" for a response from the prover, and therefore V_2 should request the next P_2 symbol.

At this point, it may be the case that some sims are waiting in (communication) states that flip a public coin, whereas some other sims are in communication states that do not flip a coin. Since V_2 is designed to present the same public coin sequence to every sim, it performs a coin flip (which ends the present stage) only if *all* the waiting sims are in coin-flipping states. Otherwise, V_2 flips no coin during its request to P_2; in this case, the sims waiting at coin-flipping states will be suspended in the subsequent segments of this stage, until the occurrence of the flip.

In response to V_2's request, P_2 is supposed to provide a 2^r-tuple whose ith element is the symbol that P_1 would send to S_i if S_i is a waiting sim that V_2 is going to continue simulating in the next segment, and a filler symbol otherwise. (Since V_2 employs no private coins, P_2 has complete information about all of the sims' configurations, as well as all the communication symbols any sim would have transmitted had it been interacting with V_1, at this point.) V_2 performs some additional controls (to be detailed below under separate headings) upon receipt of this symbol, rejecting if it detects a "cheating" attempt by P_2 at this juncture.

After the single coin flip at the end of each stage, V_2 calls the TICK subroutine, described in Section 2.3.[(vi)] If the timer does not run out at this tick, simulation proceeds with the next stage. If the timer does run out, V_2 considers the overall simulation finished.

When the overall simulation is finished, V_2 flips r more coins to select one of the sims. (Essentially, where V_2 chooses a single computation branch randomly in private during its execution, V_3 simulates all those 2^r branches publicly and then chooses one of them at the very end.) If the selected sim has been determined to have rejected, or to have entered an infinite loop, V_2 *rejects*. Otherwise, it *accepts*.

Members of L are accepted by V_2 with sufficiently high probability. Consider an intermediate verifier V_1', which is obtained by modifying V_1 so that it calls the same TICK subroutine that we mentioned above in V_2's description after every actual public coin toss of V_1 to determine whether it should time out and accept. Since the only difference between V_1 and V_1' is that V_1' may accept some input strings (which can get involved in very long computations that are cut off by the timer) with higher probability than V_1, P_1 is clearly able to convince V_1' to accept any member of L with probability at least $1 - \varepsilon_1$.[(vii)] We claim that P_2 will be able to convince V_2 to accept any member of L with the same probability, by simply obeying the protocol described above and transmitting precisely the symbols that P_1 would have transmitted in response to communications from V_1 at every step: In this case, P_2's interaction with V_2 would be a perfectly faithful imitation of P_1's interaction with V_1', with identical probabilities of acceptance, determined by the public coins flipped during the simulations, and the r final public random bits that stand for the r private coin flips of V_1'.

P_2 cannot trick V_2 into looping forever within a stage. We have already noted that each sim can have at most $f(n)$ different configurations, for some polynomial f. Consider the 2^r-tuple consisting of the configurations of all the sims, which we will refer to as the "collective configuration". Clearly, the

(vi) V_2 flips polynomially many additional public coins for each such subroutine call. P_2 waits for the subroutine to finish and the next stage of V_2's execution to resume.

(vii) Strictly speaking, V_1' faces a slightly different prover that imitates P_1 faithfully, pausing this procedure only to wait for the coin tosses of the TICK subroutines to be completed.

number of different possible collective configurations is $g(n) \in O\big(f(n)^{2^r}\big)$, which is another polynomial in n. Considering the collective configurations at the end of each segment, a stage which has continued for more than $g(n)$ segments must have therefore repeated an end-of-segment collective configuration. Note that, within any particular stage of the execution of V_2, P_2 has complete, deterministic control of the evolution of the collective configuration between subsequent communication steps, through the responses it sends. This would give a malevolent prover a chance to trick the verifier into looping forever within a stage. Since any "honest" prover-verifier interaction that eventually terminates with the verifier reaching a halting state can be shortened to avoid all such collective configuration repetitions without changing the outcome, V_2 expects P_2 to respect this convention, and concludes any stage that is seen to contain more than $g(n)$ segments by labeling any active sims that have not reached a coin-flipping state by that time as looping.

P_2 will be caught if it does not mimic P_1. As noted above, P_2 has complete information about all the sims that it is communicating with. This is quite different than the situation of P_1, which does not see the private bits of the verifier. It is sometimes possible for P_1 to deduce information about the private random bits used by V_1 by noting the communication symbols that it transmits during the interaction, but P_1 has no way of distinguishing two probabilistic branches of V_1 corresponding to distinct private coin sequences that have the same communication transcript, *i.e.*, that have both communicated exactly the same string of symbols, punctuated with public coin flips in identical locations, up to the present point. This presents a devious P_2 an opportunity to attempt to achieve a lower rejection probability than P_1 by transmitting different messages to sims that would be indistinguishable to (and would therefore have to receive the same message from) P_1. As a countermeasure, V_2 maintains an up-to-date list of groups of sims that should be presently indistinguishable to P_1, and *rejects* whenever it sees that the P_2 symbol is trying to convey different messages to different sims in the same group. As a result, any interaction where P_2 attempts to send messages that are not "P_1-compliant" in this sense leads to rejection with probability 1.

Double exponential runtime is enough for V_2 to reject non-members of L with sufficiently high probability. It remains to analyze the probability that V_2 fails to reject a non-member of L. Since P_2 is unable to make V_2 loop forever, and any attempt to deviate from the protocol described above would be caught, all we need to focus on is the additional error possibly introduced by V_2 cutting off sims with very long computations, and regarding them as having accepted. Fortunately, Condon and Lipton have shown [4, 6] that the execution of a verifier whose space consumption is logarithmically bounded can be cut off after a double-exponentially large number of steps without a significant change in the acceptance and rejection probabilities.[(viii)] By the reasoning in the proof of Theorem 6.1 of [6], for any desired small positive value of ε_x, there exists a polynomial p such that the log-space verifier V_1 will have rejected any non-member input string of length n with probability at least $1 - \varepsilon_1 - \varepsilon_x$ within at most $2^{2^{p(n)}}$ computational steps. By tuning the parameters (Section 2.3, Appendix C) of the double exponential timer, we can therefore set the error bound of V_2 to any desired value in the interval $(\varepsilon_1, 1/2)$.

(viii) The verifier model in [6] uses only private coins. The validity of this result for our verifiers is a simple consequence of the argument in the proof of Lemma 7.

We conclude the proof by noting that the language L verified by V_2 is in P, due to the fact that

$$\mathrm{IP}(\text{log-space}, \infty\text{-public-coins}, \infty\text{-time}) = \mathrm{P},$$

which was proven by Condon in [3]. □

Since it is obvious that

$$\mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, \text{poly}^{\%}\text{-public-coins}, \text{poly}^{\%}\text{-time}) \subseteq \mathrm{IP}(\text{log-space}, \text{con-private-coins}, \infty\text{-public-coins}, \infty\text{-time}),$$

Lemma 12 concludes the proof of Theorem 9.

5 Concluding remarks

We end with some remaining open questions on finite-state verifiers.

Focusing on verifiers which halt with probability 1, Dwork and Stockmeyer [9] proved that the class $\mathrm{IP}(\text{con-space}, \text{poly}_{\%}\text{-private-coins}, \text{poly}_{\%}\text{-time})$ contains all context-free languages and some NP-complete languages, but whether $\mathrm{P} \subseteq \mathrm{IP}(\text{con-space}, \text{poly}_{\%}\text{-private-coins}, \text{poly}_{\%}\text{-time})$ or not is still an open question.[(ix)] On the other hand, our Theorem 9 establishes that polynomial-time finite-state machines (even those with our additional restrictions on private-coin usage) which are allowed to be tricked into looping with an arbitrarily small probability have the capability to verify all languages in P. Does imposing the requirement that a constant-space machine should halt with probability 1 preclude it from verifying some languages?

It has been proven [14] that every language verifiable with arbitrarily low error by a polynomial-time verifier (with no bound imposed on the space consumption) is also verifiable with such a verifier that has perfect completeness. We do not know whether a counterpart of this result holds for finite-state verifiers. (Note that the construction of Lemma 10 creates verifiers without perfect completeness.)

It would be interesting to characterize the classes that are associated by reducing the public coin budgets in Section 4, like $\mathrm{IP}_*(\text{con-space}, \text{con-private-coins}, \log^{\%}\text{-public-coins}, \infty\text{-time})$. Does Condon and Ladner's result showing that logarithmic-space verifiers that flip only logarithmically many public coins cannot verify any language outside the class LOGCFL [5] have a counterpart for the constant-space case?

We also note that the following question, posed more than 30 years ago by Dwork and Stockmeyer [9], is still open:

Is there a nonregular language in $\mathrm{IP}(\text{con-space}, \text{poly}_{\%}\text{-public-coins}, \text{poly}_{\%}\text{-time})$?

Acknowledgments

We thank the user with the pseudonym *obscurans* at *Mathematics Stack Exchange* for their help with the analysis in the proof of Lemma 2. We are grateful to the anonymous referees for their helpful comments.

(ix) Dwork and Stockmeyer do construct finite-state verifiers that halt with probability 1 for all languages in $\mathrm{DTIME}(2^{O(n)})$, but those verifiers have double exponential runtime. [9]

References

[1] A. Ambainis and J. Watrous. Two-way finite automata with quantum and classical states. *Theoretical Computer Science*, 287(1):299–311, 2002.

[2] L. Babai. Trading group theory for randomness. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 421–429, New York, NY, USA, 1985. Association for Computing Machinery.

[3] A. Condon. *Computational Models of Games*. MIT Press, 1989.

[4] A. Condon. The complexity of space bounded interactive proof systems. In *Complexity Theory: Current Research*, pages 147–189. Cambridge University Press, USA, 1993.

[5] A. Condon and R. Ladner. Interactive proof systems with polynomially bounded strategies. *Journal of Computer and System Sciences*, 50(3):506–518, 1995.

[6] A. Condon and R. J. Lipton. On the complexity of space bounded interactive proofs. In *30th Annual Symposium on Foundations of Computer Science*, pages 462–467, 1989. (Extended version: `https://www.cs.ubc.ca/~condon/papers/condon-lipton89.pdf`).

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[8] H. G. Demirci, A. C. C. Say, and A. Yakaryılmaz. The complexity of debate checking. *Theory of Computing Systems*, 57(1):36–80, July 2015.

[9] C. Dwork and L. Stockmeyer. Finite state verifiers I: The power of interaction. *J. ACM*, 39(4): 800–828, Oct. 1992.

[10] L. Fortnow and C. Lund. Interactive proofs and alternating time-space complexity. *Theoretical Computer Science*, 113:55–73, 1993.

[11] M. U. Gezer and A. C. C. Say. Constant-space, constant-randomness verifiers with arbitrarily small error. *Information and Computation*, 288:104744, 2022.

[12] M. U. Gezer and A. C. C. Say. Finite state verifiers with both private and public coins. In G. Castiglione and M. Sciortino, editors, *Proceedings of the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13–15, 2023*, volume 3587 of *CEUR Workshop Proceedings*, pages 241–253. CEUR-WS.org, 2023.

[13] M. U. Gezer, Ö. Dolu, N. Ersoy, and A. C. C. Say. Real-time, constant-space, constant-randomness verifiers. *Theoretical Computer Science*, 976:114155, 2023.

[14] O. Goldreich, Y. Mansour, and M. Sipser. Interactive proof systems: Provers that never fail and random selection. In *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, pages 449–461, 1987.

[15] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 59–68. Association for Computing Machinery, 1986.

[16] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[17] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4), Sept. 2015.

[18] J. Hartmanis. On non-determinancy in simple computing devices. *Acta Informatica*, 1(4):336–344, 1972.

[19] J. Kemeny and J. Snell. *Finite Markov Chains*. Van Nostrand, 1960.

[20] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown automata. In *Proceedings of 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 92–106. IEEE Computer Society, Oct. 1978.

[21] A. C. C. Say and A. Yakaryılmaz. Finite state verifiers with constant randomness. *Logical Methods in Computer Science*, 10(3), Aug. 2014.

[22] A. Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, Oct. 1992.

Appendix

A Proof of Theorem 1

Theorem 1. *Any language recognized by a Turing machine that uses at most* $\lfloor \log(n+2) \rfloor$ *space with a work tape alphabet of size at most* 2^c *(for some integer constant* $c > 0$*) and within* $t(n)$ *time can also be recognized by a* $(c+5)$*-head finite automaton within* $t(n) \cdot ((2c + 3/2)n + 2c + 2)$ *time.*

Proof: Let M be any logarithmic-space Turing machine as described in the theorem statement, and L be its language. Without loss of generality, let the work tape alphabet of M be the set of c-digit binary numbers from 0 to $2^c - 1$, where 0 corresponds to the blank symbol of the alphabet.

We will construct a $(c+5)$-head finite automaton R, whose heads will be denoted $h_1, \ldots, h_c$, h_α, h_β, h_γ, h_δ, and h_ϵ. R is to recognize L by simulating M. The locations (indices) of heads h_1 through h_c will encode the contents of M's work tape, with the ith head's index matching the number whose $\lfloor \log(n+2) \rfloor$-bit binary representation is written in reverse on the ith track, *i.e.*, the sequence obtained by concatenating the ith bit of each symbol of the work tape string. (The leftmost input tape cell containing $\triangleright$ has index 0.) h_α will mimic M's input tape head exactly, and h_β will mimic the distance of M's work tape head from the left end in its position, albeit on the input tape of R. The remaining three heads will be used as auxiliaries to facilitate the manipulations of the indices of h_1 through h_c required for the simulation of M's memory.

We now recall some multihead automaton programming techniques that will be used in the machine described below. Our algorithms sometimes require a head to walk from its original position to the left end of the tape in order to control the number of iterations of a loop, or to perform calculations based on information encoded in its index. This original index position may also be needed for future operations. For instance, since the content of the ith track is represented by the index of h_i in R, one has to make sure that this information is stored elsewhere when h_i needs to take such a walk. This is achieved by making some auxiliary head h_a (that is originally at position 0) mirror h_i, *i.e.*, h_a moves right whenever h_i moves to the left during this procedure. For ease of reference, one can then simply swap the identities of h_a and h_i to restore h_i's index to its original value before the walk. This mirroring method can be modified easily to double or halve the index of a given head: Starting with h_i at index x and h_a at index 0, move h_a twice as fast (resp. slow) as h_i, so that h_a arrives at index $2x$ (resp. $\lfloor x/2 \rfloor$) when h_i arrives at the left end. In the algorithms to be presented below, we will use a separate font to denote these DOUBLE and HALVE procedures as a reminder that they involve loops, and therefore take more than unit time to execute, although they are specified in a single line of the pseudocode.

R will simulate each transition of M using a sequence of transitions of its own, and report the same decision as M. Each transition of M consists of two parts:

1. "Sensing" by the current configuration; which involves taking account of the internal state of the machine and reading the symbols under both the input and work tape heads.
2. "Acting" on the current configuration; which can involve changing the internal state, the symbol under the work tape head, and the positions of both the input and work tape heads.

The rest of the proof describes the simulation of these parts.

Sensing. R traces M's current state in its own finite memory and reads its input tape with h_α to obtain the symbol M is scanning with its input head.

Obtaining the symbol under M's work tape head requires sophistication. Let b denote the index of h_β, which can range from 0 (when h_β is on $\triangleright$) up to $\lfloor \log(n+2) \rfloor - 1$ (since M uses at most $\lfloor \log(n+2) \rfloor$ work tape cells). To decode the symbol under M's work tape head from the positions of heads h_1 through h_c, essentially, the least significant b bits of their indices will be truncated and the last bits of what remains will be retrieved. This is performed sequentially for each head, although a machine with more auxiliary heads could perform this in parallel to expedite the sensing routine. In Algorithm 3, the loop controlled by Stage 1.1 walks h_i to the left so that its original index j_i is reduced to $\lfloor j_i / 2^b \rfloor$ at the end of this walk. Each iteration of this loop runs for half as many steps as the previous one, while performing the HALVE procedure described above. Since the runtime of the following parity check stage (1.2) is just $\lfloor j_i / 2^b \rfloor$, we conclude that "reading" the symbol under the simulated work tape head with this procedure incurs at most

$$\sum_{i=1}^{c} \Big(\underbrace{j_i + \lfloor \tfrac{j_i}{2} \rfloor + \lfloor \tfrac{j_i}{4} \rfloor + \cdots + \lfloor \tfrac{j_i}{2^{b-1}} \rfloor}_{\text{Stage 1.1}} + \underbrace{\lfloor \tfrac{j_i}{2^b} \rfloor}_{\text{Stage 1.2}} \Big) \leq \sum_{i=1}^{c} 2j_i \leq 2cn + 2c$$

steps.

The reader should note that the execution of the procedure of Algorithm 3 leaves $h_1, \ldots, h_c$, and h_β (after the reidentifications) at their original locations. We will justify the statement at Stage 0 of Algorithm 3 by the end of this proof.

0. (At this point, h_γ, h_δ, and h_ϵ are guaranteed to be on $\triangleright$.)
1. Do the following for $i = 1, \ldots, c$:
1.1. Walk h_β to $\triangleright$, using h_γ to store h_β's original index, and repeating the following parallelly with every leftward move of h_β:
1.1.1. HALVE the index of h_i (from x to $\lfloor x/2 \rfloor$), using h_δ as auxiliary. In the first execution of this stage (*i.e.*, the first iteration of loop 1.1), use h_ϵ as auxiliary to parallelly store the original index of h_i. } $O(n)$ steps
1.2. Determine the parity of h_i's index by walking h_i to $\triangleright$. If the parity is even, "read" the ith bit of the symbol under M's simulated work tape head as 0. Otherwise, read it as 1. } $O(n)$ steps
1.3. Reidentify the auxiliary heads h_ϵ and h_γ to set h_i and h_β back to their locations at the beginning of the procedure.

Algorithm 3: Using R's heads to read a symbol on the work tape of M.

Acting. Updating the internal state and head positions of the simulated machine is again straightforward: R updates the simulated machine's state in its own finite memory. It also moves h_α and h_β, which mimic M's input and work tape heads, in the required directions, after simulating the changes on M's work tape contents at that step.

Changes on the work tape are accomplished as follows. Once again, let b denote the index of h_β, ranging from 0 to $\lfloor \log(n+2) \rfloor - 1$. If the bit on the ith track of the simulated work tape symbol at that index needs to be changed from 1 to 0, h_i is moved 2^b cells to the left. If that bit is to change from 0 to 1, h_i is moved 2^b cells to the right. In Algorithm 4, Stages 1 and 2 have the combined effect of moving h_ϵ to

the 2^bth index on the input tape. The exponentiation is achieved by repeated doubling. The loop controlled by Stage 4 then updates all relevant head indices by 2^b units in the required directions parallelly.

The loop controlled by Stage 2 is iterated up to $\lfloor \log(n+2) \rfloor - 1$ times, with each iteration taking twice as much time as its predecessor. The runtime of Stage 4 equals 2^b, which is not greater than $n/2 + 1$. The procedure therefore incurs up to

$$\underbrace{1}_{\text{Stg. 1}} + \underbrace{2 + 4 + 8 + \cdots + 2^b}_{\text{Stage 2}} + \underbrace{2^b}_{\text{Stg. 4}} \leq 3 \cdot 2^b - 1 \leq 3n/2 + 2$$

steps.[x] The task of updating the simulated machine state and the simulated input and work heads' positions would incur a single step by themselves, and are handled in parallel with the procedure of Algorithm 4, requiring no additional time.

0. (At this point, h_γ, h_δ, and h_ϵ are on $\triangleright$.)	
1. Move h_ϵ one step to the right.	$O(n)$ steps
2. Walk h_β to $\triangleright$, using h_γ to store h_β's original index, and repeating the following parallelly with every leftward move of h_β:	
2.1. DOUBLE the index of h_ϵ, using h_δ as auxiliary.	
3. Reidentify the auxiliary head h_γ to set h_β back to its location at the beginning of the procedure.	
4. Walk h_ϵ to $\triangleright$, repeating the following parallelly with every leftward move of h_ϵ:	$O(n)$ steps
4.1. If the bit on the ith track should change from 0 to 1, move h_i one step to the right. If the bit on the ith track should change from 1 to 0, move h_i one step to the left. All such heads move simultaneously in the same transition.	

Algorithm 4: Rearranging R's heads to simulate a change on the worktape of M.

The two routines for reading and changing the simulated memory contents are carried out back to back, and they both leave the heads h_γ, h_δ, and h_ϵ on the left end-marker as they finish. Moreover, these heads are on the left end-marker also when R begins its execution, by definition. The assumptions in Stage 0 of both routines are thus justified.

The two parts together take at most $(2c + 3/2)n + 2c + 2$ steps, which is an upper bound to the multiplicative runtime overhead of R simulating M. Thus, R completes its simulation within $t(n) \cdot ((2c + 3/2)n + 2c + 2)$ steps. □

B Proof of Lemma 2

Lemma 2. *For any integer $t > 0$, integer-valued function $f(n) \in O(n^t)$, and desired "error" bound $\varepsilon_{\text{premature}} > 0$, there exists a probabilistic finite automaton with expected runtime in $O\left(n^{t+1}\right)$, such that the probability that this machine halts in fewer than $f(n)$ time-steps is $\varepsilon_{\text{premature}}$.*

Proof: By the implications of $f(n) \in O(n^t)$, let $c > 0$ and $n_0 > \varepsilon_{\text{premature}}^{-1}$ be integers large enough to satisfy $f(n) \leq c \cdot n^t$ for all $n \geq n_0$. The extra constraint upon n_0 to be larger than $\varepsilon_{\text{premature}}^{-1}$ will be

[x] Note that, when $b = 0$, *i.e.*, when h_β is scanning $\triangleright$, R can proceed to Stage 4 directly without performing any transition associated with Stage 2. ("Stage 3" always incurs 0 steps.)

evident by the end of the proof. We will analyze the runtime requirements of each stage of the probabilistic finite automaton M_{t,c,n_0} presented in Algorithm 5.

1. If $n < n_0$, pause for $f(n)$ time-steps and *halt*.
2. If $t = 1$, move the input head back to $\triangleright$. Then, move the input head right until it reaches $\triangleleft$, while pausing it for $c - 1$ time-steps after each movement, then *halt*.
3. Repeat the following t times:
3.1. Move the input head to the beginning of the input string.
3.2. Perform a random walk with the input head (*i.e.*, repeatedly move the input head towards left or right by the flips of a coin) until the input head reaches $\triangleright$ or $\triangleleft$. Pause for $c - 1$ time-steps after each step moving the input head.
4. If any of the last t random walks have ended with the input head reaching $\triangleright$, go back to Stage 3.
5. *Halt*.

Algorithm 5: The probabilistic finite automaton M_{t,c,n_0}.

Stage 1 starts with $\min(n+2, n_0+1)$ steps to check whether $n < n_0$. If so, M_{t,c,n_0} runs for a total of $n + 2 + f(n)$ time-steps and then halts, which yields a runtime within desired bounds with certainty.

When $n \geq n_0$ and $t = 1$, M_{t,c,n_0} halts after running for $2n_0 + cn + 2$ steps in Stages 1 and 2 combined, which is again within the expected and minimum time limits, and with certainty.

The rest of the proof will examine the claims for $n \geq n_0$ and $t > 1$.

Expected runtime. Each random walk at Stage 3.2 is commonly known as the *gambler's ruin*, and the probability of each ending at the right-hand side of the input tape is $(n+1)^{-1}$ by their Markov chain analysis [19, Section 7.1]. Consequently, the probability of having a *terminating batch* (*i.e.*, a batch of t random walks where each of them ends up at the right-hand side of the input tape) is $(n+1)^{-t}$.

The expected number of independent trials to observe a binomial event is the reciprocal of its occurrence probability [7, Section C.4]. Therefore, the expected number of batches until termination is $(n+1)^t$, and the same for random walks is $t \cdot (n+1)^t$.

The expected runtime of each random walk is cn steps [19, Section 7.1]. The reset in Stage 3.1 takes exactly $n_0 + 2$ steps before the first random walk, and it is expected to take

$$\underbrace{(n+1)^{-1} \cdot (n+2)}_{\text{resetting from } \triangleleft} + \underbrace{(1 - (n+1)^{-1}) \cdot 1}_{\text{resetting from } \triangleright} = 2$$

steps before every subsequent one. Taking Stage 1 into account as well, the expected runtime of M_{t,c,n_0} is

$$t \cdot (n+1)^t \cdot (cn+2) + 2n_0 + 1 \in O\big(n^{t+1}\big).$$

Minimum runtime. For simplicity, we note that $(n+1)^{-t}$, the probability of a terminating batch occurring, is less than n^{-t}.

It is evident that the execution of a terminating batch takes more than tnc time-steps, and thus, in a time frame of $f(n) \leq c \cdot n^t$ steps, there are at most $\frac{n^{t-1}}{t}$ opportunities for terminating batches to occur, which is less than n^{t-1} (given $t > 1$).

The probability of an event with an occurrence probability less than n^{-t} happening at least once given less than n^{t-1} opportunities, is less than

$$g_t(n) = 1 - \left(1 - n^{-t}\right)^{n^{t-1}}.$$

Consider the following binomial expansion:

$$\begin{aligned} g_t(n) &= 1 - \left(\sum_{k=0}^{n^{t-1}} (-1)^k \cdot \binom{n^{t-1}}{k} \cdot n^{-tk} \right) \\ &= \frac{n^{t-1}}{n^t} - \frac{n^{t-1}}{n^t} \cdot \frac{n^{t-1}-1}{2n^t} + \frac{n^{t-1}}{n^t} \cdot \frac{n^{t-1}-1}{2n^t} \cdot \frac{n^{t-1}-2}{3n^t} - \cdots \end{aligned}$$

Since the terms are alternating and the magnitudes are decreasing, we have

$$g_t(n) < \frac{n^{t-1}}{n^t} = n^{-1}.$$

$g_t(n)$ constitutes an upper bound to the "error" of M_{t,c,n_0} halting sooner than the desired lower bound of $f(n)$ time-steps. Since $g_t(n)$ is monotone decreasing for positive n, $g_t(\bar{n}) < \varepsilon_{\text{premature}}$ follows for all $\bar{n} \geq n_0$ (recalling that $n_0 > \varepsilon_{\text{premature}}^{-1}$). With Stage 1 ensuring that M_{t,c,n_0} waits long enough for the strings shorter than n_0 with probability 1, the probability of M_{t,c,n_0} halting prematurely will be less than $\varepsilon_{\text{premature}}$ for strings of any length. Note that $\varepsilon_{\text{premature}}$ can be set to any desired small positive value by simply employing a suitably large number of states (to make n_0 appropriately large) in the implementation of Stage 1. □

C The double exponential timer

Algorithm 6 restates the subroutine TICK described in Section 2.3. Recall that we also maintain a global Boolean variable T that tracks whether all the red coins flipped since the last reset of this variable came out heads. The integers c and t are parameters that we can use for setting the expected number of ticks. Stage 2 uses logarithmic space in the work tape to count the coin tosses.

1. Toss a red coin and update the Boolean variable T according to the outcome.
2. Toss $p(n) = c \cdot n^t$ blue coins.
3. If any blue coin outcome at Stage 2 was tails, *return* with the timer still running.
4. (Arriving at this stage implies that all the blue coins of Stage 2 came up heads.) If T indicates that one or more red coins flipped since the last reset came out tails, reset T and *return* with the timer still running.
5. (Arriving at this stage implies that all red coins flipped at Stage 1 since the last reset of T were heads.) *Return* and announce that the timer has run out.

Algorithm 6: The TICK subroutine.

To analyze the expected amount of subroutine calls (ticks) before the timer runs out, we define the following random variables to represent the described counts:

X: Times the subroutine is called.

Y: Times the tick advances control to Stage 4, *i.e.*, all $p(n)$ blue coins come up heads.

Z: Number of red coins flipped since the last reset as the tick advances to Stage 4 (at any given time).

Z_i: Number of red coins flipped since the last reset as the tick advances to Stage 4 for the ith time.

Let $\mathbb{E}[O]$ denote the expected value of a random variable O. The expected value that we are after is $\mathbb{E}[X]$.

The probability that any tick advances to Stage 4 (*i.e.*, $p(n)$ blue coins come up heads) is $2^{-p(n)}$. Hence, we expect there to be $2^{p(n)}$ attempts, each one flipping another red coin, since the last reset of variable T before control advances to Stage 4 in some subroutine call. Thus, we have

$$\mathbb{E}[Z] = 2^{p(n)}.$$

We can say the following for X, trivially by the fact that each tick flips exactly one red coin at its Stage 1:

$$X = Z_1 + Z_2 + \cdots + Z_Y.$$

Since the expected value operator is a linear operator, we have

$$\mathbb{E}[X] = \mathbb{E}[Z_1] + \mathbb{E}[Z_2] + \cdots + \mathbb{E}[Z_Y].$$

Since each $\mathbb{E}[Z_i]$ trivially equals $\mathbb{E}[Z]$, for a particular $Y = y$, we have

$$\mathbb{E}[X \mid Y = y] = 2^{p(n)} \cdot y,$$

and by the Law of Total Expectation, we have

$$\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]] = \mathbb{E}\Big[2^{p(n)} \cdot Y\Big] = 2^{p(n)} \cdot \mathbb{E}[Y].$$

To determine $\mathbb{E}[Y]$, consider a subroutine call in which control has just advanced to Stage 4. Let z denote the number of red coins flipped since the last reset if the program had reached Stage 4 before, and since the beginning of the program's run otherwise. Such a subroutine call will advance further to Stage 5, indicating time-out, if all z coin outcomes were heads. The probability of this is 2^{-z}. If we had $Z = z$ at any given time Stage 4 was reached, then we would expect Stage 5 to be reached after Stage 4 having been reached the following number of times:

$$\mathbb{E}[Y \mid Z = z] = 2^z.$$

Then, by the Law of Total Expectation, we have

$$\mathbb{E}[Y] = \mathbb{E}[\mathbb{E}[Y \mid Z]] = \mathbb{E}\big[2^Z\big].$$

By Jensen's Inequality, we have

$$\mathbb{E}\big[2^Z\big] \geq 2^{\mathbb{E}[Z]}.$$

If we multiply both sides with $2^{p(n)}$ and then make the substitutions by the equalities we established, we obtain our lower bound:

$$\begin{aligned} 2^{p(n)} \cdot \mathbb{E}\left[2^Z\right] &\geq 2^{p(n)} \cdot 2^{\mathbb{E}[Z]}, \\ 2^{p(n)} \cdot \mathbb{E}[Y] &\geq 2^{p(n)} \cdot 2^{2^{p(n)}}, \\ \mathbb{E}[X] &\geq 2^{p(n)} \cdot 2^{2^{p(n)}}. \end{aligned}$$

This proves that the expected number of subroutine calls is in $2^{2^{\Omega(n^t)}}$.

D Construction of $2\mathrm{afa}\ M$ in the proof of Theorem 6

Given a constant-space verifier $V = (Q_V, \Sigma, \Phi, \Gamma, \delta_V, q_{V,0})$ which flips at most r public coins, we will construct a 2afa M as described in the proof of Theorem 6. Let Q_{pub}, Q_{com}, and $\{\, q_{V,\mathrm{acc}}, q_{V,\mathrm{rej}} \,\}$ be the subsets of Q_V that correspond to public coin flips, communications, and halting, respectively, as described in Section 2.1. Since V can simulate a constant-space work tape in its finite state, we will assume that V repeatedly writes $\sqcup$ with its work tape head without ever moving it, without loss of generality.

For any given set of coin-flip sequences S, where each such sequence is a string in $\{\, 0, 1 \,\}^*$ representing the outcomes of a fair coin, let $\mathit{PREFIX}(S)$ (resp. $\mathit{PROPER\text{-}PREFIX}(S)$) be the set of all prefixes (resp. proper prefixes) of sequences in S.

Let $\mathit{BIN}_{\leq r}$ denote the set of all coin-flip sequences up to length r and let MAJ_r be the set of all prefix-free subsets of $\mathit{BIN}_{\leq r}$ which have total probability greater than $1/2$, *i.e.*,

$$\begin{aligned} \mathit{BIN}_{\leq r} &= \left\{\, x \;\middle|\; x \in \{\, 0,1 \,\}^*, |x| \leq r \,\right\}, \\ \mathit{MAJ}_r &= \left\{\, S \;\middle|\; S \subseteq \mathit{BIN}_{\leq r}, S \cap \mathit{PROPER\text{-}PREFIX}(S) = \emptyset, \textstyle\sum_{x \in S} 2^{-|x|} > 1/2 \,\right\}. \end{aligned}$$

Let $M = (Q, \Sigma, \delta, q_0)$. We define Q as the union of four mutually exclusive sets.

$$Q = \{\, q_0 \,\} \cup Q_{\mathrm{main}} \cup Q_{\mathrm{flip}} \cup \{\, q_{\mathrm{acc}}, q_{\mathrm{rej}} \,\},$$

where

$$\begin{aligned} Q_{\mathrm{main}} &= \{\, (S, q_V, \gamma, p) \mid S \in \mathit{MAJ}_r, q_V \in Q_V, \gamma \in \Gamma, p \in \mathit{PREFIX}(S) \,\}, \\ Q_{\mathrm{flip}} &= \{\, (S, q_V, \gamma, p, b_{\mathrm{pub}}) \mid S \in \mathit{MAJ}_r, q_V \in Q_{\mathrm{pub}}, \gamma \in \Gamma, p \cdot b_{\mathrm{pub}} \in \mathit{PREFIX}(S), b_{\mathrm{pub}} \in \{\, 0,1 \,\} \,\}, \end{aligned}$$

and q_{acc} and q_{rej} are the accepting and rejecting states, respectively.

The set of universal states, $Q_\forall$, is the following subset of Q_{main}:

$$Q_\forall = \{\, (S, q_V, \gamma, p) \mid S \in \mathit{MAJ}_r, q_V \in Q_{\mathrm{pub}}, \gamma \in \Gamma, p \in \mathit{PROPER\text{-}PREFIX}(S) \,\}.$$

All other non-halting states of M are existential.

At every step, the first four components of the states from Q_{main} and Q_{flip} will keep the following information:

1. A set of coin sequences that are supposed to lead V's simulation to acceptance, which has been guessed by the initial transition and will remain constant for the rest of the execution.

2. The simulated state of V.
3. The content of the simulated communication cell.
4. The history of coin outcomes used in the simulation up to this point. (This component will initially contain the empty string, λ.)

The states in $Q_\forall$ and Q_{flip} will be used in a two-step procedure to execute a universal branching (to simulate both possible outcomes of a coin toss of V parallelly when necessary), immediately followed by an existential branching (to simulate the prover response to V), with the last component of states in Q_{flip} holding that coin outcome during the procedure, as will be detailed below.

We now present the transition function of M in parts. For all $q \in Q \setminus \{ q_{\text{acc}}, q_{\text{rej}} \}$ and $\sigma \in \Sigma_{\rhd\lhd}$, $\delta(q, \sigma)$ has the value $\{ q_{\text{rej}} \}$, unless specified otherwise below.

The initial transition. The first step of M, which guesses a set S from MAJ_r, is realized by

$$\delta(q_0, \rhd) = \{ ((S, q_{V,0}, \sqcup, \lambda), 0) \mid S \in MAJ_r \}.$$

Transitions simulating the non-communicating transitions of V. We have

$$\delta((S, q_V, \gamma, p), \sigma) = \{ ((S, q'_V, \gamma, p), d_i) \} \text{ where } \delta_V(q_V, \sigma, \sqcup, \gamma) = (q'_V, \sqcup, d_i, 0),$$

for all $((S, q_V, \gamma, p), \sigma) \in Q_{\text{main}} \times \Sigma_{\rhd\lhd}$, where q_V is from $\overline{Q_{\text{com}}} \setminus \{ q_{V,\text{acc}}, q_{V,\text{rej}} \}$.

Transitions simulating the communicating transitions of V that do not flip a coin. We have

$$\delta((S, q_V, \gamma, p), \sigma) = \{ ((S, q'_V, \gamma'', p), d_i) \mid \gamma'' \in \Gamma, \delta_V(q_V, \sigma, \sqcup, \gamma) = (q'_V, \sqcup, \gamma', d_i, 0) \}$$

for all $((S, q_V, \gamma, p), \sigma) \in Q_{\text{main}} \times \Sigma_{\rhd\lhd}$, where q_V is from $\overline{Q_{\text{pub}}} \cap Q_{\text{com}}$. Here, we note that the content of the simulated communication cell switches into γ'' (the response from the simulated prover) directly, without ever becoming γ' (the communication from V). This transition is existential, meaning that M will reach acceptance if and only if there is a symbol $\gamma'' \in \Gamma$ that, when communicated to V, can lead V to acceptance.

Transitions simulating a coin toss and the ensuing communication. As previously mentioned, V's transitions involving a coin toss are simulated by M in two consecutive transitions. The first one is to ensure that all the coin flip sequences in S are covered universally:

$$\delta((S, q_V, \gamma, p), \sigma) = \{ ((S, q_V, \gamma, p, b_{\text{pub}}), 0) \mid b_{\text{pub}} \in \{ 0, 1 \}, p \cdot b_{\text{pub}} \in PREFIX(S) \}$$

for all $((S, q_V, \gamma, p), \sigma) \in Q_\forall \times \Sigma_{\rhd\lhd}$. Note that this transition merely introduces the simulated coin outcome into the state, turning it into a member of Q_{flip}.

Following that first transition, the second transition finishes simulating the interaction of V with its prover:

$$\delta((S, q_V, \gamma, p, b_{\text{pub}}), \sigma) = \{ ((S, q'_V, \gamma'', p \cdot b_{\text{pub}}), d_i) \mid \gamma'' \in \Gamma, \delta_V(q_V, \sigma, \sqcup, \gamma, b_{\text{pub}}) = (q'_V, \sqcup, \gamma', d_i, 0) \}$$

for all $((S, q_V, \gamma, p, b_{\text{pub}}), \sigma) \in Q_{\text{flip}} \times \Sigma_{\rhd\lhd}$. Since this transition is existential, every prover response $\gamma'' \in \Gamma$ is considered, as described for other transitions from communicating states of V above. The coin

value b_{pub} used in the present branch is added to the history of coin flips carried in the fourth component of the 2afa state.

Note that states of the form (S, q_V, γ, p) such that $q_V \in Q_{\mathrm{pub}}$ and $p \in S$ are directed to *reject* immediately, since they correspond to a scenario where the verifier tosses an additional coin that is not covered by the set S.

Halting. If the simulation of V reaches a halting state, M also halts with the same decision in its next transition, *i.e.*,

$$\delta((S, q_V, \gamma, p), \sigma) = \begin{cases} \{\, q_{\mathrm{acc}} \,\} & \text{if } q_V = q_{V,\mathrm{acc}}, \\ \{\, q_{\mathrm{rej}} \,\} & \text{if } q_V = q_{V,\mathrm{rej}}, \end{cases}$$

for all $((S, q_V, \gamma, p), \sigma) \in Q_{\mathrm{main}} \times \Sigma_{\rhd\lhd}$ where q_V is either $q_{V,\mathrm{acc}}$ or $q_{V,\mathrm{rej}}$.

E Proof of Lemma 7

Lemma 7. *The following is true for all functions s, f, g, and t:*

$$\mathrm{IP}_*(O(s(n))\text{-space}, O(f(n))\text{-private-coins}, O(g(n))\text{-public-coins}, O(t(n))\text{-time}) \subseteq \mathrm{IP}_*(O(s(n))\text{-space}, O(f(n)+g(n))\text{-private-coins}, O(t(n))\text{-time}).$$

This inclusion also holds for the $\mathrm{IP}(\ldots)$ *variants of the classes, and when worst-case time bounds are replaced with those corresponding to expected usage.*

Proof: Let $V_1 = (Q_1, \Sigma, \Phi, \Gamma_1, \delta_1, q_0)$ be a verifier that uses $O(f(n))$ private and $O(g(n))$ public random bits for some functions f and g. We construct a verifier $V_2 = (Q_2, \Sigma, \Phi, \Gamma_2, \delta_2, q_0)$ that uses $O(f(n) + g(n))$ private random bits and no public coins. The protocol that V_2 engages in with the prover is almost identical to that of V_1, with the exception that V_1's public coin outcomes are emulated by private random bits of V_2 that are sent through the communication cell.

Q_2 is a superset of Q_1, where all the states that flip public coins in V_1 have been rebranded as states that flip only private coins in V_2. V_2's program inherits all transitions of V_1 emanating from states that do not flip public coins. V_2 simulates each transition of V_1 from a state q that involves such a flip as described below.

If q does not flip a private coin, the V_1 transition (cf. Table 1)

$$\delta_1(q, \sigma, \phi, \gamma, b_{\mathrm{pub}}) = (q', \phi', \gamma', d_i, d_w)$$

is replaced in V_2 by the transition

$$\delta_2(q, \sigma, \phi, \gamma, b_{\mathrm{pri}}) = (q', \phi', (\gamma', b_{\mathrm{pri}}), d_i, d_w),$$

making use of an additional symbol in Γ_2 that conveys both the original communication symbol γ' and the simulated public coin outcome in a single package. Note that q remains a communication state in V_2 in this case.

If, on the other hand, q flips a private coin as well, the corresponding V_1 transition

$$\delta_1(q, \sigma, \phi, \gamma, b_{\mathrm{pri}}, b_{\mathrm{pub}}) = (q', \phi', \gamma', d_i, d_w)$$

is simulated by two V_2 transitions

$$\delta_2(q, \sigma, \phi, \gamma, b_{\mathrm{pri},1}) = ((q, b_{\mathrm{pri},1}), \phi, 0, 0),$$
$$\delta_2((q, b_{\mathrm{pri},1}), \sigma, \phi, \gamma, b_{\mathrm{pri},2}) = (q', \phi', (\gamma', b_{\mathrm{pri},2}), d_i, d_w),$$

that flip two private coins, utilize new states in Q_2 to store the first coin outcome temporarily, and then send the required information through the communication cell. In this case, q becomes a non-communication state in V_2. The two new states $(q, 0)$ and $(q, 1)$ (introduced for each such q) are branded as communication states that also flip (only) private coins.

For each public coin flip removed from V_2's transition function (and thereby its execution), exactly one private coin flip is introduced. V_2 emulates V_1, operating with the same verification error. Its runtime is at most double that of V_1. V_2 uses the same amount of work tape as V_1. □