Finite State Verifiers with Both Private and Public Coins

M. Utkan Gezer[1,*], A. C. Cem Say[1]

[1]*Department of Computer Engineering, Boğaziçi University, Bebek 34342, İstanbul, Türkiye*

Abstract

We consider the effects of allowing a finite state verifier in an interactive proof system to use a bounded number of private coins, in addition to "public" coins whose outcomes are visible to the prover. Although swapping between private and public-coin machines does not change the class of verifiable languages when the verifiers are given reasonably large time and space bounds, this distinction has well known effects for the capabilities of constant space verifiers. We show that a constant private-coin "budget" (independent of the length of the input) increases the power of public-coin interactive proofs with finite state verifiers considerably, and provide a new characterization of the complexity class P as the set of languages that are verifiable by such machines with arbitrarily small error in expected polynomial time.

Keywords

Interactive proof systems, Delegating computation, Verifiable computing

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 [1]. These systems involve a computationally weak "verifier" (a probabilistic Turing machine with small resource bounds) engaging in a dialogue with a very strong but possibly malicious "prover", 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 "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 a communication channel. "Public-coin" systems, on the other hand, hide nothing from the prover, who can

Proceedings of the 24th Italian Conference on Theoretical Computer Science, Palermo, Italy, September 13–15, 2023

*Corresponding author.

utkan.gezer@boun.edu.tr (M. U. Gezer); say@boun.edu.tr (A. C. C. Say)

be assumed to observe the coin flips and deduce the resulting changes to the configuration of the verifier as they unfold. It is known [2] 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 [3].

In this paper, we study the capabilities of constant-space verifiers (essentially, two-way probabilistic finite automata) which are allowed to hide some, but not necessarily all, of their coin flips from the prover. We show that allowing these machines to use even only a constant number[1] of private coins enlarges the class of verified languages considerably. We adapt several previous results from the literature to this framework and present a new characterization of the complexity class P as the set of languages that can be verified by such machines in polynomial time with arbitrarily small error.

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, 4].

A *verifier* in an *interactive proof system (IPS)* is a 6-tuple $(Q, \Sigma, \Phi, \Gamma, \delta, q_0)$, where

1. Q is the finite set of states, such that $Q = Q_{\text{pri}} \cup Q_{\text{pub}} \cup \{ q_{\text{acc}}, q_{\text{rej}} \}$ where
   - Q_{pri} is the set of states that flip private coins,
   - Q_{pub} is the set of states that flip public coins, $Q_{\text{pri}} \subseteq Q_{\text{pub}}$, and
   - $q_{\text{acc}}, q_{\text{rej}} \notin Q_{\text{pri}} \cup Q_{\text{pub}}$ are the accept and reject states, respectively,
2. Σ is the input alphabet,
3. Φ is the work tape alphabet, which is guaranteed to include the special blank symbol, $\sqcup$,
4. Γ is the communication alphabet, $\sqcup \in \Gamma$,
5. δ is the transition function, described below, and
6. q_0 is the initial state, $q_0 \in Q$.

The computation of a verifier is initialized with $\triangleright w \triangleleft$ written on its read-only input tape, where $w \in \Sigma^*$ is the input string, and $\triangleright, \triangleleft \notin \Sigma$ are the left and right end-markers, respectively. The input head of the verifier is initially on the left end-marker. The read-write work tape is initially filled with blank symbols, with the work tape head positioned at the beginning of the tape. Apart from these two tapes, the verifier also has access to a communication cell, which is a single-cell tape that is initially blank.

Let $\Sigma_{\bowtie} = \Sigma \cup \{ \triangleright, \triangleleft \}$. 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".

[1] That is, the number of such private coin flips is fixed, regardless of the length of the input.

The computation of a verifier evolves by its transition function δ, which is constructed in two parts as follows: For $q \in Q_{\text{pri}}$ (which implies $q \in Q_{\text{pub}}$), $\delta(q, \sigma, \phi, \gamma, b_{\text{pri}}, b_{\text{pub}}) = (q', \phi', \gamma', d_i, d_w)$ dictates that if the machine is originally in state q, scanning $\sigma \in \Sigma_{\rhd\lhd}$ in the input tape, $\phi \in \Phi$ in the work tape, and $\gamma \in \Gamma$ in the communication cell, and has obtained the "private" random bit $b_{\text{pri}} \in \{ 0, 1 \}$ and the "public" random bit $b_{\text{pub}} \in \{ 0, 1 \}$ as the result of two independent fair coin flips, then it will switch to state $q' \in Q$, write $\phi' \in \Phi$ to the work tape, overwrite the communication cell with γ', and move the input and work tape heads in the directions $d_i, d_w \in \Delta$ respectively. For $q \in Q_{\text{pub}} \setminus Q_{\text{pri}}$, $\delta(q, \sigma, \phi, \gamma, b_{\text{pub}}) = (q', \phi', \gamma', d_i, d_w)$ dictates a similar transition in which the outcome of only a single public coin is used.

The verifier is paired with another entity, the prover, whose aim is to convince the verifier to accept (or to prevent it from rejecting) its input string. At every step of the verifier's execution, the outcome of the public coin flip is automatically communicated to the prover. The prover can be modeled as a function that determines the symbol $\gamma \in \Gamma$ which will be written in the communication cell in between the transitions of the verifier based on the input string, the public coin outcomes, and the sequence of symbols written by the verifier to the communication cell up to that point. Note that the prover does not "see" (and, in the general case, cannot precisely deduce) the configuration of a verifier which uses private coins.

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}. Note that the verifier may 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 $\varepsilon^+, \varepsilon^- < 1/3$ where

- for all input strings $w \in L$, there exists a prover P such that V halts by accepting with probability at least $1 - \varepsilon^+$ when started on w and interacting with P, and,
- for all input strings $w \notin L$ and for all provers P^*, 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. Intuitively, this definition requires V (in order for it to keep its error low) to be reasonable enough to accept legitimate arguments that prove that the input is a member of the language in question, yet skeptical enough to reject spurious claims of membership when the input is not in the language, both with high probability, even when it is interacting with the most cunning of all provers.

In some of our proofs in Section 3, we will be considering 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. Sections 2.2 and 2.3 provide more information on automata with multiple input heads and their relationships with the standard Turing machine model.

We will be using the notation $\text{IP}(\text{resource}_1, \text{resource}_2, \ldots, \text{resource}_k)$ to denote the class of languages that can be verified with arbitrarily small (but possibly positive) errors by machines that operate within the resource bounds indicated in the parentheses. These may represent

budgets for runtime, working memory 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 "poly-time" or "$\log$-space". 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.[2]

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. Some of the interactive protocols to be discussed have the property that the verifier has a probability ε of being fooled to run forever by a malicious prover trying to prevent it from rejecting the input. The designer of the protocol can reduce ε to any desired small positive value. The denotation "*" will be used to mark that the indicated amount corresponds to such a machine's expected consumption of a specific resource with the remaining (high) probability $1 - \varepsilon$. For instance, "poly^*-time" will indicate that the verifier's expected runtime is polynomially bounded with probability almost, but possibly not exactly, 1.

2.2. Multihead finite automata and finite state verifiers

Our work makes use of an interesting relationship [5] between multihead finite automata and logarithmic-space Turing machines, which will be detailed in Section 2.3. In this subsection, we provide the necessary definitions and establish the link between these machines and probabilistic finite state verifiers.

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. $\delta\colon Q \times \Sigma_{\bowtie}^k \to \mathcal{P}\big(Q \times \Delta^k\big)$ 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 whether or not to move each head, given the machine's current state and the list of symbols that are currently being scanned by the k input heads, and $\Sigma_{\bowtie}$ 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 $\triangleright w \triangleleft$ written on its tape, and all k of its heads on the left end-marker. At each step, M

[2] Verifiers that use only some private coins and no public coins can be described in the framework given above by simply specifying their transition functions to be insensitive to the value of the public random bit argument, i.e., $\delta(q, \sigma, \phi, \gamma, b_{\mathrm{pri}}, 0) = \delta(q, \sigma, \phi, \gamma, b_{\mathrm{pri}}, 1)$ and $\delta(q, \sigma, \phi, \gamma, 0) = \delta(q, \sigma, \phi, \gamma, 1)$ for all values of q, σ, ϕ, γ, and b_{pri}.

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 final 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.

Our main result will be making use of a technique introduced by Say and Yakaryılmaz [4] for "simulating" a multihead nondeterministic automaton in an interactive proof system whose verifier is a (single-head) probabilistic finite automaton. This method's application to the problem studied in this paper will be explained in detail in the proof of Lemma 6 in Section 3.

2.3. Multihead finite automata and logarithmic space Turing machines

The equivalence of multiple input heads and logarithmic amounts of memory was discovered by Hartmanis [5]. The following theorem reiterates this result in detail, and also contains an analysis for the overhead in time incurred during the simulation.

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

Proof idea. Assume, for the sake of simplicity, that n is a power of 2. A string over the alphabet $\{ 0, 1 \}$ in a work tape of length $\log n$ can be seen as the digits of a number between 0 and $n - 1$, inclusive, represented in binary. The index of a head on the input tape can similarly range between 0 (when on $\rhd$) and $n + 1$ (when on $\lhd$). This correspondence enables a multihead finite automaton to store the same information there is on a $\log n$ symbol string over an alphabet of size 2^c (which can be viewed as the binary digits of c numbers stacked on top of each other), encoded at the indices of c input heads.

There is a way for a multihead finite automaton to retrieve a single binary digit of a head's index and also to change it. Four spare input heads are introduced and used to accomplish these functions, with one of them mimicking the position of the simulated work tape head, and the other three helping with the index manipulations for decoding, changing, and then re-encoding. One last input head is the input head of the multihead finite automaton.

Using this method, a multihead finite automaton can simulate a $\lceil \log n \rceil$-space Turing machine directly.

The detailed proof can be found in [6, Appendix A.1].

2.4. Implementing a polynomial-time "clock" in a probabilistic finite automaton

A logarithmic-space Turing machine can "clock" its own execution to satisfy any desired polynomial time bound by counting up till that bound in the logarithmic space available. The constant-space machines we construct in Section 3 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 constant $t > 0$, integer-valued function $f(n) \in O(n^t)$, and desired "error" bound $\varepsilon_{premature} > 0$, there exists a probabilistic finite automaton with an expected runtime in $O(n^{t+1})$, such that the probability that this machine halts in fewer than $f(n)$ time-steps is $\varepsilon_{premature}$.*

Proof idea. Assume, for the sake of simplicity, that t is an integer. We program a probabilistic finite state automaton to make t random walks with its input head, each starting from the first symbol on the input and ending at either one of the end-markers. If all the walks have ended on the right end-marker, the machine halts. Otherwise, the process is repeated. The analysis shows that such a machine has all the necessary characteristics in its runtime.

The detailed proof can be found in [6, Appendix A.2].

3. Finite state verifiers with constant private randomness

Let us consider the language of palindromes, $L_{\text{pal}} = \{\, w \in \{\, 0, 1 \,\}^* \mid w = w^R \,\}$, where x^R is the reverse any string x. Trivially, $L_{\text{pal}} \in \mathcal{L}(\text{2nfa}(*), \text{linear-time})$.

We recall the following facts about the power of finite state verifiers at the two extreme ends of the "public vs. private" spectrum, which shows us that even a finite amount of private coins gives verifiers an edge that no amount of public coins can compensate:

Fact 3. $L_{\text{pal}} \notin \text{IP}(\text{con-space}, \infty\text{-public-coins}, \infty\text{-time})$ [2].

Fact 4. $\mathcal{L}(\text{2nfa}(*), \text{linear-time}) \subseteq \text{IP}(\text{con-space}, \text{con-private-coins}, \text{linear}^*\text{-time})$ [7].[3]

Let us now examine the effects of allowing finite state verifiers to hide a constant number of their coin flips from the prover. This 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 5.

$$\text{IP}(\text{con-space}, \text{con-private-coins}, \text{poly}^*\text{-public-coins}, \text{poly}^*\text{-time}) = \text{P}.$$

[3] Recall from the definition of our IP complexity class notation in Section 2 that the verifier's runtime can be infinite with probability at most ε, and its expected runtime is bounded as indicated with the remaining large probability.

Proof. It is known [8, 1] that

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

The proof follows from this fact and Lemmas 6 and 7. □

Lemma 6.

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

More specifically, for any $t > 1$,

$$\mathrm{IP}(\text{log-space}, O(n^t)\text{-public-coins}, O(n^t)\text{-time}) \subseteq \mathrm{IP}(\text{con-space}, \text{con-private-coins}, O(n^{t+2})^*\text{-public-coins}, O(n^{t+2})^*\text{-time}).$$

Proof. For some $t > 1$, let V_1 be a public-coin verifier that uses $O(\log n)$ space and $O(n^t)$ time to verify the language L with error $\varepsilon_1 > 0$. We will assume that the work tape of V_1 is exactly $\lceil \log n \rceil$ cells long, but with a "multi-track" alphabet (e.g., as in [5]) to accommodate for the required amount of memory.

In the following discussion, let any prover facing V_1 be called P_1. Since V_1 cannot be fooled into accepting a non-member of L with high probability no matter what prover it is facing, it is also immune against any such P_1 that "knows" V_1's algorithm. Since all coins are public, such a P_1 can be assumed to have complete knowledge about V_1's configuration at every step of their interaction. Therefore we will assume that V_1 sends no further information through the communication cell without loss of generality.

Let us consider 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 means of Theorem 1. Since the simulation is direct and does not involve any additional use of randomness, V_2 recognizes L with the same probability of error ε_1. The only time overhead is caused by the simulation of the log-space memory, so, by Theorem 1, V_2 will complete its execution in $O(n^{t+1})$ time. Just like V_1, V_2 sends no information to its prover, say, P_2, except the outcomes of its public coins.

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 every step) to verify L.

V_3 performs the following verification for m consecutive rounds:

First, V_3 flips r of its private coins. Thanks to this randomness, it picks the ith head of V_2 with some probability[4] p_i. 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. In this process, V_3 traces the selected head of V_2 with its own single head, and relies on the messages of P_3 to inform it about what the other heads of V_2 would be reading at any step of the execution. V_3 does not send any information (except, of course, the outcomes of its public coins) to P_3. P_3, on the other hand, is expected to transmit both

[4]We will discuss constraints on these values in the discussion below.

- what P_2 would be transmitting to V_2, and
- its claims about the readings of all k heads of V_2

at every step of the simulated interaction. V_3 verifies the part of these claims regarding the head it had chosen in private, and *rejects* if it sees any discrepancy. P_3 sends a special symbol when it claims that the simulated interaction up to that point has ended with V_2 reaching acceptance. If this is consistent with what V_3 has been able to validate, and if this was not the mth round, V_3 proceeds to another round.

While p_i is positive for all i, the sum $p_{\text{simulation}} = \sum_{i=1}^{k} p_i$ is very small by design. With the remaining high probability $p_{\text{timer}} = 1 - p_{\text{simulation}}$, V_3 passes this round operating as a probabilistic timer that has an expected runtime of $O\left(n^{t+2}\right)$. This timer is also guaranteed to run longer than 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 P_3 claims that V_2 has accepted before the timer runs out, then V_3 proceeds with another round of verification. Otherwise (if the timer runs out before P_3 declares acceptance), 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 analyzes the error and runtime of V_3.

Arbitrarily small verification error. For any input string that is a member of L, P_3 should tell V_3 the truth about what V_2 would read with its k heads, and emit the messages P_2 would send to V_2 alongside those readings. Faced with such a truthful P_3, V_3 may erroneously reject at any given round, either due to the simulated V_2 also rejecting,[5] or due to a premature timeout of the probabilistic timer. The probability of that is $p_{\text{simulation}} \cdot \varepsilon_1 + p_{\text{timer}} \cdot \varepsilon_{\text{premature}}$. For V_3 to accept such an input string, it should go through m consecutive rounds of verification without committing such errors. The probability that V_3 will fail to accept a string in L is therefore

$$\varepsilon_3^+ \leq 1 - \left(1 - \left(p_{\text{simulation}} \cdot \varepsilon_1 + p_{\text{timer}} \cdot \varepsilon_{\text{premature}}\right)\right)^m.$$

For any input not in L, V_3 can accept only if P_3 claims that V_2 accepts in all m rounds. Such a claim can either be true, since V_2 can genuinely accept such a string with probability at most ε_1; or false, in which case P_3 would be "lying", i.e., providing false information that could be detected when compared against the actual readings of at least one of V_2's heads. Let $p_{\min} = \min_{i=1}^{k} p_i$. The probability of V_3 failing to catch such a lie in any round is at most $1 - p_{\min}$. It follows that the probability that V_3 accepts a string not in L is

$$\varepsilon_3^- \leq \max(\varepsilon_1, (1 - p_{\min}))^m.$$

The last kind of verification error for V_3 is getting tricked into running forever by an evil prover when given a non-member input string. The probabilistic timer function, when in play with probability p_{timer}, will keep V_3 from running forever. The probability of V_3 looping on the ith round of its verification is at most $\max(\varepsilon_1, (1 - p_{\min}))^{i-1} \cdot p_{\text{simulation}}$ (since it should pass

[5] Our definitions allow V_2 to reject members of L with some small probability.

the first $i-1$ rounds without rejecting in that case). The probability that V_3 can be fooled to loop is at most the sum of those probabilities, i.e.,

$$\varepsilon_3^{\text{loop}} \leq p_{\text{simulation}} \cdot \sum_{i=0}^{m-1} \max(\varepsilon_1, (1-p_{\min}))^i.$$

The overall error bound of V_3 is the maximum of all three, i.e.,

$$\varepsilon_3 = \max\left(\varepsilon_3^+, \varepsilon_3^-, \varepsilon_3^{\text{loop}}\right).$$

Since all of these bounds can be lowered arbitrarily to any positive constant (by first increasing m to constrain ε_3^-, and then decreasing $p_{\text{simulation}} > 0$ and $\varepsilon_{\text{premature}} > 0$ to constrain the other two), ε_3 can also be lowered to any desired positive constant.

Polynomial expected runtime with arbitrarily high probability. With $\varepsilon_3^{\text{loop}}$ set to a desired small 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 $O\left(n^{t+1}\right)$ time or will operate as the probabilistic timer that has the expected runtime of $O\left(n^{t+2}\right)$. Thus, it is expected to run in $O\left(n^{t+2}\right)$ time with arbitrarily high probability. □

Lemma 7.

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

More specifically, for any integer $t > 1$,

$$\text{IP}(\text{con-space}, \text{con-private-coins}, O\left(n^t\right)^*\text{-public-coins}, O\left(n^t\right)^*\text{-time}) \subseteq \text{IP}(\text{log-space}, O\left(n^{t+1}\right)\text{-public-coins}, O\left(n^{t+1}\right)\text{-time}).$$

Proof. Let V_1 be a (single-head) constant space verifier that uses at most r private coins and an unlimited budget of public coins to verify a language L, for some constant r. The three types of errors that V_1 may commit are that

- it might reject a member of L when communicating with an honest prover with some probability ε_1^+,
- it might be tricked to accept a non-member of L with some probability ε_1^-,
- it might be tricked to run forever when the input is not a member of L with some probability $\varepsilon_1^{\text{loop}}$.

When it is not running forever (i.e., with probability $1-\varepsilon_1^{\text{loop}}$), V_1 is expected to terminate in $f_1(n) \in O\left(n^t\right)$ steps where $t > 1$ is an integer. In the following, the prover that V_1 interacts with will be named P_1.

A constant space public-coin (2^r+t)-head verifier V_2 can verify L in polynomial time and with an error bound close to that of V_1 as follows: V_2 will run 2^r parallel simulations ("sims") of

V_1, where the ith sim S_i (for $i \in \{ 0, \ldots, 2^r - 1 \}$) assumes its private random bits as the bits of the binary representation of the number i and uses the $(i+1)$st head of V_2. The prover that V_2 interacts with, which we name P_2, is supposed to mimic P_1 by providing a 2^r-tuple containing the symbols that P_1 would send to each S_i at each step. At every step of its interaction, V_2 performs the following three tasks:

- It checks the communication symbol received from P_2 to see if it is consistent with the simulated interaction that took place up to that point between P_1 and the sims (as will be detailed below), rejecting otherwise.
- It updates the simulated state information and moves the head corresponding to each sim in accordance with V_1's transition function, the latest public coin outcome, the input symbol scanned by the corresponding head and the communication symbol received from P_2 addressed to that sim.
- It sends P_2 a 2^r-tuple containing the communication symbols emitted by all the sims at the present step.

The consistency check mentioned above is necessary for the following reason: Consider two distinct sims which correspond to two probabilistic paths that emit precisely the same sequence of communication symbols up to a certain point during an interaction of V_1 with P_1. Since P_1 is unable to determine which of these two paths it is talking to at that point, it cannot send different communication symbols to these sims. V_2 is supposed to check that P_2 respects this condition, and never sends different symbols to two sims whose communications have been identical since the beginning of the interaction. V_2 can keep track of subsets of such similar-looking sims in its finite memory to implement this control at every step.

V_2 uses its remaining t heads to implement a deterministic clock that runs for $f_2(n) = cn^t$ steps (where c is a positive integer whose value ensures that $f_2(n) \gg f_1(n)$, and determines the error committed by V_2, as will be described below) in the background.[6] V_2 makes its decision when the clock times out by picking one of the sims at random with equal probability by the result of r public coin tosses. It accepts if the chosen sim has accepted on time, and rejects otherwise.

V_2 is not able to carry out a "perfect" simulation of V_1 (with identical acceptance and rejection probabilities) because of the strict bound on its runtime. This causes V_2's decisions to differ from those of V_1 in the following two ways:

1. V_1 has a nonzero probability of accepting some inputs after running for more than $f_2(n)$ steps, whereas V_2 rejects in branches of its simulation corresponding to such cases.
2. V_2 rejects and halts on each branch of its simulation corresponding to cases where V_1 is tricked to run forever.

Let ε_2^+ and ε_2^- be the counterparts in V_2 of the errors ε_1^+ and ε_1^-, respectively. (V_2 can not be fooled into looping, so we do not have to worry about that type of error.) Let us analyze how the two differences described above affect the errors of V_2 compared to those of V_1.

[6] See [7, Lemma 3] for an explanation of how such a clock can be constructed for any desired value of c.

ε_2^- is at most ε_1^-, since none of the differences between V_2 and V_1 can cause an increase in an acceptance probability.

ε_2^+ is greater than ε_1^+ by the probability that V_1 runs longer than $f_2(n)$ steps and then accepts. By definition, the expected runtime of V_1 is $f_1(n)$ when it is not running forever (i.e., with probability $1 - \varepsilon_1^{\text{loop}}$). By Markov's inequality, the probability that V_1 runs for more than $f_2(n)$ steps when it is not running forever is at most $f_1(n)/f_2(n)$. This difference can be reduced by increasing c, thus bringing ε_2^+ arbitrarily close to ε_1^+, and proving our claim that the overall error bound of V_2 is close to that of V_1.

We will conclude the proof by demonstrating V_3, a standard public coin log-space verifier with a single input head that verifies the same language. The naive way of simulating a multi-head machine by a logarithmic space machine is rather straightforward. Specifically, V_3 can keep V_2's head indices in multiple tracks of its work tape in binary format. (To accommodate for $2^r + t$ tracks in the work tape, V_3 should use a work tape alphabet of size 2^{2^r+t}.) In each simulated transition of V_2, to decipher what V_2 is reading with its $2^r + t$ heads, V_3 will carry out the following steps:

1. Move the input head to $\triangleright$.
2. Do the following for all $i \in \{ 1, \ldots, 2^r + t \}$:
3. Decrement the index on the ith track of the work tape and move the input head to the right. Repeat this until the index becomes 0.
4. Register the symbol under the input head as x_i.
5. Increment the index on the ith track of the work tape and move the input head to the left. Repeat this until the head is reading $\triangleright$.

Having learned the symbols $x_1, \ldots, x_{2^r+t}$ scanned by the simulated V_2's heads, V_3 can use the latest public coin flip and consult the communication cell to complete a simulated transition of V_2, updating the work tape contents to reflect the new head positions of V_2 by incrementing or decrementing the indices on the respective tracks. Note that the matching prover, P_3, which is supposed to send the messages that P_2 would be sending for each simulated step, will send "filler" symbols (all of which will be ignored by V_3) through the communication cell while it waits for V_3 to complete these walks on its work tape. V_3 accepts the input only if it is convinced that V_2 accepts the same as a result of this interaction.

V_3's runtime is simply the runtime of V_2 multiplied by the overhead of simulating multiple input heads within the logarithmic work tape. Counting from 0 to n (or down from n to 0) in binary takes $O(n)$ time for a Turing machine by amortized analysis. Incrementing or decrementing binary numbers with $\lceil \log n \rceil$ digits takes $O(\log n)$ time. As a result, using the naive method of simulation explained above, V_3 is expected to run in $O\left(n^{t+1}\right)$ time. □

The runtime of V_3 in Lemma 7 can be improved by introducing $2^r + t$ logarithmically-long caches in the memory, one for each head of the simulated V_2, each containing the slice from the input string where the corresponding head resides at that time. This slightly more advanced way of simulating multiple heads using logarithmic space (which has been previously used and demonstrated in detail in [7]) saves V_3 a factor of $O(\log n)$ in runtime, but we stuck with the naive method for its sufficiency and simplicity.

Theorem 8.

$$\mathrm{NC} \subseteq \mathrm{IP}\big(\text{con-space, con-private-coins, } O(n^4)^*\text{-public-coins, } O(n^4)^*\text{-time}\big).$$

Proof. It is known [9] that

$$\mathrm{NC} \subseteq \mathrm{IP}\big(\text{log-space, 0-private-coins, } O(n \log^2 n)\text{-public-coins, } O(n \log^2 n)\text{-time}\big).$$

Since $\log^2 n \in O(n)$ by standard asymptotic analysis, we also have

$$\mathrm{NC} \subseteq \mathrm{IP}\big(\text{log-space, 0-private-coins, } O(n^2)\text{-public-coins, } O(n^2)\text{-time}\big).$$

The claimed result then follows directly from Lemma 6. □

4. Concluding remarks

This line of research can be expanded with various further questions. Although we mentioned the effect of cutting off the usage of public coins completely (Fact 4), we did not consider the results of imposing a tight budget on the number of public coins that the verifier can flip. (The "clock" head's random walk in Section 2.4 has an expected cost of polynomially many such flips.) It would be interesting, for instance, to ask whether Condon and Ladner's result stating that logarithmic space verifiers that flip only logarithmically many public coins can not verify any language outside the class LOGCFL [10] has a counterpart for the constant space case or not.

Acknowledgments

The authors thank the anonymous referees for their comments. This research was partially supported by Boğaziçi University Research Fund Grant Number 19441. Utkan Gezer's participation in this work is supported by the Turkish Directorate of Strategy and Budget under the TAM Project number 2007K12-873.

References

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

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

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

[4] A. C. C. Say, A. Yakaryılmaz, Finite state verifiers with constant randomness, Logical Methods in Computer Science 10 (2014).

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

[6] M. U. Gezer, A. C. C. Say, Finite state verifiers with both private and public coins, arXiv e-prints (2023). arXiv:2306.09542.
[7] M. U. Gezer, A. C. C. Say, Constant-space, constant-randomness verifiers with arbitrarily small error, Information and Computation 288 (2022) 104744.
[8] A. Condon, Computational Models of Games, MIT Press, 1989.
[9] L. Fortnow, C. Lund, Interactive proofs and alternating time-space complexity, Theoretical Computer Science 113 (1993) 55–73.
[10] A. Condon, R. Ladner, Interactive proof systems with polynomially bounded strategies, Journal of Computer and System Sciences 50 (1995) 506–518.