Real-Time, Constant-Space, Constant-Randomness Verifiers

Özdeniz Dolu¹, Nevzat Ersoy¹, M. Utkan Gezer¹, and A. C. Cem Say¹

Department of Computer Engineering, Boğaziçi University, Bebek 34342, İstanbul, Turkey
{ozdeniz.dolu,nevzat.ersoy,utkan.gezer,say}@boun.edu.tr

Abstract. We study the class of languages that have membership proofs which can be verified by real-time finite-state machines using only a constant number of random bits, regardless of the size of their inputs. Since any further restriction on the verifiers would preclude the verification of nonregular languages, this is the tightest computational budget which allows the checking of externally provided proofs to have meaningful use. We show that all languages that can be recognized by two-head one-way deterministic finite automata have such membership proofs. For any $k > 0$, there exist languages that cannot be recognized by any $k$-head one-way nondeterministic finite automaton, but that are nonetheless real-time verifiable in this sense. The set of nonpalindromes, which cannot be recognized by any one-way multihead deterministic finite automaton, is also demonstrated to be verifiable within these restrictions.

Keywords: Interactive Proof Systems · Real-time finite automata · Probabilistic finite automata

1 Introduction

The characterization of problem classes in terms of the computational requirements on machines that are supposed to check purported proofs of membership of their input strings in a language has been an important theme of complexity theory, leading to landmark achievements like the PCP Theorem and celebrated open questions like the $P$ vs. $NP$ problem.

As expected, imposing tighter bounds on the computational resources of the verifiers for these proofs of membership seems to restrict the associated language classes: Limiting a polynomial-time deterministic verifier to use only a logarithmic, rather than polynomially bounded amount of working memory “shrinks” the class of verifiable languages to $NL$ from $NP$, and the same apparent loss of power also occurs when a logarithmic-space, polynomial-time probabilistic verifier is restricted to use only a constant, rather than logarithmically bounded number of random bits for bounded-error verification.

In this paper, we focus on the tightest possible “budgetary” restrictions that can be imposed on such verifiers by considering the case where the machine’s
working memory and the amount of usable random bits are both constants irrespective of the input length, and the runtime is maximally constrained, so that only a real-time scan of the input string is allowed. We examine the class of languages whose membership proofs can be checked under these extreme conditions. Note that decreasing the number of random bits from a positive constant to zero would make such a proof system equivalent to a nondeterministic finite automaton, unable to recognize any nonregular languages. Since membership in any regular language can be decided by a “stand-alone” real-time deterministic finite automaton with no need of an externally provided certificate, the machines we consider are truly the weakest possible verifiers of meaningful use, highlighting the issues involved in the checking of the proofs of extremely long claims.

We build on previous work [12] which showed an equivalence between constant-space, constant-randomness verifiers and multihead nondeterministic finite automata working as language recognizers. This equivalence breaks down when the machines are restricted to consume their inputs in real-time fashion: A real-time multihead automaton is no more powerful than a single-head one, and can only recognize a regular language, whereas Say and Yakaryılmaz were able to demonstrate a nonregular language [12] which has membership proofs that can be checked by a real-time finite-state verifier with a fixed number of coin tosses.

In this paper, we prove the following facts about these very weak verifiers: All languages that can be recognized by two-head one-way deterministic finite automata have membership proofs that can be verified by these machines. For any \( k > 0 \), there exist languages that cannot be recognized by any \( k \)-head one-way nondeterministic finite automaton, but that are nonetheless real-time verifiable in this sense. The set of nonpalindromes, which cannot be recognized by any one-way multihead deterministic finite automaton, is also demonstrated to be verifiable in this setup. We conjecture that the real-time requirement truly decreases the verification power, i.e. that there exist languages that can be verified only when the definition of these machines is relaxed to allow them the ability to pause on the input tape.

The rest of the paper is structured as follows: Section 2 provides the necessary definitions and previous results regarding the relation between multihead finite automata and constant-randomness finite-state verifiers. Our results are presented in Section 3. Section 4 is a conclusion.

2 Preliminaries

2.1 One-way multihead finite automata

A one-way \( k \)-head nondeterministic finite automaton (1nfa(\( k \))) is a nondeterministic finite-state machine with \( k \) read-only heads that it can direct on an input string flanked by two end-marker symbols. Each head can be made to stay put or move one symbol to the right in each computational step. Formally, a 1nfa(\( k \)) is a 6-tuple \((Q, \Sigma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})\), where

1. \( Q \) is the finite set of internal states,
2. $\Sigma$ is the finite input alphabet,
3. $\delta: Q \times \Sigma^k \rightarrow \mathcal{P}(Q \times \Delta)$ 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:
   - $\Delta = \{0, +1\}$ is the set of possible head movements, where 0 means “stay put” and +1 means “move right”;
   - $\Sigma_{end} = \Sigma \cup \{\triangleright, \triangleleft\}$, where $\triangleright, \triangleleft \notin \Sigma$ are respectively the left and right end-markers, placed automatically to mark the boundaries of the input,
4. $q_0 \in Q$ is the initial state,
5. $q_{acc} \in Q$ is the final state at which the machine halts and accepts, and
6. $q_{rej} \in Q$ is the final state at which the machine halts and rejects.

Given an input string $w \in \Sigma^*$, a $\text{1nfa}(k)$ $M = (Q, \Sigma, \delta, q_0, q_{acc}, q_{rej})$ 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 timestep, $M$ 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 the right end-marker.

Each different sequence of choices $M$ may take corresponds to a different computation history, i.e. a sequence of tuples describing all the state and head positions that $M$ goes through in that particular eventuality.

$M$ is said to accept $w$ if there exists a computation history where it reaches the state $q_{acc}$, given $w$ as the input. $M$ is said to reject $w$ if every computation history of $M$ on $w$ either reaches $q_{rej}$, ends with a transition whose associated set of choices is $\emptyset$, or a head has moved beyond the right 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.

A one-way $k$-head deterministic finite automaton, denoted $\text{1dfa}(k)$, is a special case of $\text{1nfa}(k)$ $(Q, \Sigma, \delta, q_0, q_{acc}, q_{rej})$ whose transition function presents exactly one “choice” of move for every input ($|\delta(q, x_1, \ldots, x_k)| = 1$ for all $q \in Q$ and $x_1, \ldots, x_k \in \Sigma_{end}$).

$\text{1nfa}(1)$ and $\text{1dfa}(1)$ are simply called one-way nondeterministic and deterministic finite automata, respectively. The “real-time” versions of these single-head machines are obtained by forcing the head to move to the right at each step (by setting $\Delta = \{+1\}$). Real-time nondeterministic and deterministic finite automata have runtimes of at most $n + 2$ on input strings of length $n$.

The classes of languages recognized by each of the machine models defined above will be denoted by the uppercase versions of the associated machine denotations. For example, $\text{1NFA}(6)$ denotes the class of languages recognizable by $\text{1nfa}(6)$’s. The following facts [7] about these language classes will be useful:

For any $k \geq 1$,

$$\text{1DFA}(k) \subset \text{1DFA}(k + 1).$$

$$\text{1NFA}(k) \subset \text{1NFA}(k + 1).$$
\( L_{\text{nonpal}} \) is the language which contains every string except palindromes on the alphabet \( \{0, 1\} \). This language can be recognized by a \( 1\text{NFA}(2) \). Since its complement cannot be recognized by any \( 1\text{DFA}(k) \) for any \( k \), there exists no deterministic one-way multihead automaton that recognizes \( L_{\text{nonpal}} \) either, by the fact \([11]\) that the class of languages recognized by \( 1\text{DFA}(k) \)’s is closed under complementation. This proves the inequality \( 1\text{NFA}(2) \setminus \bigcup_k 1\text{DFA}(k) \neq \emptyset \).

2.2 Verifiers

There exist several elegant characterizations of language classes in terms of bounds imposed on the resources available to probabilistic Turing machines (“verifiers”) tasked with checking purported proofs (“certificates”) of membership of their input strings in a language.

Formally, a verifier is a 6-tuple \((Q, \Sigma, \Phi, \Gamma, \delta, q_0)\), where

1. \( Q \) is the finite set of states, such that \( Q = P \cup D \cup \{q_{\text{acc}}, q_{\text{rej}}\} \) where
   - \( P \) is the set of coin-tossing states,
   - \( D \) is the set of deterministic states, such that \( P \cap D = \emptyset \), and
   - \( q_{\text{acc}} \) and \( q_{\text{rej}} \) are the accept and reject states, respectively.
2. \( \Sigma \) is the input alphabet, not containing the end-markers \( \triangleright \) and \( \triangleleft \),
3. \( \Phi \) is the work tape alphabet,
4. \( \Gamma \) is the certificate alphabet, not containing \( \triangleright \),
5. \( \delta \) is the transition function, described below, and
6. \( q_0 \) is the initial state, \( q_0 \in Q \).

As in Section 2.1, \( \Sigma_{\triangleright \triangleleft} \) will be used to denote the union \( \Sigma \cup \{\triangleright, \triangleleft\} \).

The transition function \( \delta \) is constructed in two parts, as follows: For \( q \in P \), \( q' \in Q \), \( \sigma \in \Sigma_{\triangleright \triangleleft} \), \( \phi, \phi' \in \Phi \), \( \gamma \in \Gamma \cup \{\triangleleft\} \), \( b \in \{0, 1\} \), \( d_i, d_w, d_c \in \{-1, 0, +1\} \), and \( d_c \in \{0, +1\} \), \( \delta(q, \sigma, \phi, \gamma, b) = (q', \phi', d_i, d_w, d_c) \) dictates that the machine will switch to state \( q' \), write \( \phi' \) on the work tape, and move the input, work tape and certificate heads in directions \( d_i \), \( d_w \), and \( d_c \), respectively, if it is originally in state \( q \), scanning \( \sigma \), \( \phi \), and \( \gamma \) on the three respective tapes, and has obtained the random bit \( b \) as the result of a fair coin toss. For \( q \in D \), \( \delta(q, \sigma, \phi, \gamma) = (q', \phi', d_i, d_w, d_c) \) dictates a similar, but deterministic transition.

A verifier halts with acceptance (rejection) when it executes a transition entering \( q_{\text{acc}} \) (\( q_{\text{rej}} \)). Any transition that moves the input or certificate head beyond an end-marker delimiting the string written on the associated read-only tape leads to a rejection, unless that last move enters \( q_{\text{acc}} \). The head on the certificate tape is defined to be one-way, since it is known \([11]\) that allowing two-way access to that tape can lead to “unfair” accounting of the space usage. The input and work tape heads are two-way in the general definition above, although we will be considering restricting the movement types of the input tape head (and completely removing the work tape) in most of the following.

We say that such a machine \( V \) verifies a language \( L \) with error \( \epsilon \) if there exists a number \( \epsilon < 1 \) where

- for all input strings \( w \in L \), there exists a certificate string \( c_w \) such that \( V \) halts by accepting with probability 1 when started on \( w \) and \( c_w \), and,
– for all input strings $w \not\in L$ and for all certificates $c$, $V$ halts by rejecting with probability at least $1 - \epsilon$ when started on $w$ and $c$.

We will be using the notation $\text{VER} (\text{restriction}_1, \text{restriction}_2, \ldots, \text{restriction}_k)$ to denote the class of languages that can be verified by machines that operate within the added restrictions indicated in the parentheses. These may represent bounds for runtime, working memory usage, and number of random bits to be used as a function of the length of their input strings. The terms 0, con, log, 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 “one-way” mode, where the input head is not allowed to move left, will be indicated by the parameter “1way-input”, whereas the further restriction to real-time movement, where the head is not allowed to pause at any step during its left-to-right scan, will be indicated by “rt-input”.

The following characterizations in terms of zero-error verifiers are well known.

\[
\text{VER} (\text{poly-time}, \text{poly-space}, 0\text{-random-bits}) = \text{NP} \\
\text{VER} (\text{poly-time}, \text{log-space}, 0\text{-random-bits}) = \text{NL}
\]

When one allows nonzero error, significant gains in space usage seem to be achievable:

\[
\text{VER} (\text{poly-time}, \text{log-space}, \text{log-random-bits}) = \text{NP} \quad \text{[4]} \\
\text{VER} (\text{con-space}, \text{con-random-bits}) = \text{NL} \quad \text{[12]}
\]

For verifiers using at least logarithmic space, the magnitude of the one-sided error can be reduced without significant increase in the runtime, whereas the constant-space verifiers of [12] (all of which have correct certificates that can be checked in polynomial time) do not seem [6] to have this property in general.

Say and Yakaryılmaz [12] also considered the case where a constant-space, constant-randomness verifier is forbidden to move its input head to the left. Using their techniques, one can obtain the following characterization:

**Theorem 1.**

\[
\text{VER} (\text{con-space}, \text{con-random-bits}, 1\text{-way-input}) = \bigcup_k \text{1NFA}(k)
\]

*Proof.* Given a 1Nfa($k$) $M$ recognizing a language $L_M$, one can construct a one-way, constant-space, constant-randomness verifier $V_M$ for $L_M$ as follows: $V_M$ expects the certificate to contain a proof of the existence of an accepting computation history (in the form of a sequence of tuples representing the nondeterministic branch taken and list of symbols scanned by the heads at each step) of $M$ working on the input string. $V_M$ uses its random bits to select a head of $M$ and

\footnote{Note that a constant-space machine is equivalent to a finite-state automaton with no work tape, since the bounded amount of information in the work tape of a constant-space verifier can also be kept using a suitably large set of internal states.}
simulates its execution on the input, relying on the certificate for information on what symbols would be scanned by the other heads of $M$ at every step. If $V_M$ ever sees the certificate reporting that the head it is tracking is currently scanning a symbol other than the correct value, it rejects. If the input is in $L_M$, a correct certificate that carries $V_M$ to acceptance with probability 1 exists. Otherwise, in order to trick $V_M$ to reach an accept state, the certificate would have to “lie” about what is being seen by at least one of the heads of $M$ in at least one step, and $V_M$ has a constant probability of having selected that head, and therefore rejecting the input. Since $M$ can be assumed to run in linear time in all its nondeterministic branches without loss of generality, any attempt by an overly long certificate to trick $V_M$ to loop without accepting will also be caught by nonzero probability.

In the reverse direction, given a finite-state verifier $V$ with one-way input that uses at most $r$ random bits, one can build a $\text{Infa}(2^r) M_V$ for the verified language $L_V$ as follows: $V$’s behavior on each different random bit sequence can be represented by a deterministic verifier obtained by “hardwiring” that particular sequence into $V$’s transition function. $M_V$ is designed to nondeterministically guess a certificate and use its heads to simulate all these $2^r$ deterministic verifiers operating on the input string and the common certificate. For each newly guessed certificate symbol, $M_V$ goes through all the deterministic verifiers, tracing each one’s execution (by changing its state and possibly moving the corresponding head) until that deterministic verifier accepts, rejects, or performs a transition consuming that new certificate symbol by moving its certificate tape head. (Since the collection of deterministic verifiers has only a fixed number of possible tuples of states, $M_V$ can detect when the deterministic verifiers run for more than that number of steps without moving any input heads, and reject on such nondeterministic branches corresponding to unnecessarily long certificates.) This procedure continues until either a deterministic verifier rejects, or all the $2^r$ deterministic verifiers are seen to accept. $M_V$ accepts if it arrives in a state representing all the deterministic verifiers having accepted.

This link between finite-state constant-randomness verifiers and multihead automata is broken when one further restricts the input heads to be real-time: A multihead finite automaton operating all its heads in real time is easily seen to be no stronger than a single-head finite automaton, and therefore cannot recognize a nonregular language. Say and Yakaryılmaz, however, were able to demonstrate a finite-state constant-randomness verifier with real-time input that verifies the nonregular language $L_{\text{twin}} = \{ w\#w \mid w \in \{0,1\}^* \}$ on the alphabet $\{0,1,\#\}$. The certificate is expected to consist of the string $w$, which is supposed to appear on both sides of the symbol $\#$ in the input. The machine tosses a coin to decide whether it should compare the substring appearing to the left or to the right of the $\#$ with the certificate as it is consuming the input in real time, and accepts only if this comparison is successful. Acceptance

\footnote{Note that the construction in the proof of Theorem produces a multihead automaton with heads that can pause on the input, even when it is fed a verifier with real-time input.}
with probability 1 is only possible for members of the language associated with well-formed certificates.

Note that such a machine must use its capability to pause the certificate tape head for some steps. This is easy to see when one considers the computational power of a verifier with real-time heads on both the input and certificate tapes: All the “deterministic” verifiers that can be obtained from the probabilistic verifier by hardwiring the possible random sequences (as we saw in the proof of Theorem 1) would then be running both their heads on exactly the same strings in perfect synchrony, and it would be possible to build a single real-time one-head finite automaton simulating this collection. This machine would be equivalent to a one-head nondeterministic finite automaton, with no power of recognizing nonregular languages.

In a very real sense, $\text{VER}(\text{con-space, con-randomness, rt-input})$ corresponds to the weakest computational setup where externally provided proofs are meaningful. In the next section, we will examine this interesting class and its relationship with $\text{VER}(\text{con-space, con-randomness, 1-way-input})$ in detail.

3 Real-time, finite-state, constant-randomness verification

3.1 $\text{1DFA}(2)$ is real-time verifiable

We start by demonstrating that every language that is recognizable by a $\text{1dfa}(2)$ is verifiable by a constant-space, constant-randomness verifier that scans its input in real time. The technique employed in the proof of Theorem 1 for constructing verifiers is not useful here, since it requires the verifier to pause its input head occasionally when processing certain portions of the certificate. We will show that all languages in $\text{1DFA}(2)$ have more concise proofs of membership that can be checked by our restricted machines. Some examples of languages on the alphabet $\{0, 1, \#\}$ in $\text{1DFA}(2)$ are $L_{\text{twin}}$, the set of all strings containing equal numbers of 0's and 1's, the set of all odd-length binary strings with the symbol $\#$ at the middle position, the language $\{w \mid w \in (x\#)^+, x \in (0 \cup 1)^+\}$, and their complements.

Theorem 2. $\text{1DFA}(2) \subseteq \text{VER}(\text{con-space, con-randomness, rt-input})$.

Proof. Let $M = (Q_M, \Sigma, \delta_M, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a $\text{1dfa}(2)$ recognizing some language $A$. At any given step of its execution, $M$ might be moving none, one, or both of its heads. We start by modifying $M$ to obtain a $\text{1dfa}(2)$ $M' = (Q_{M'}, \Sigma, \delta_{M'}, q_0, q_{\text{acc}}, q_{\text{rej}})$ that recognizes the same language while moving exactly one of its heads at every step, starting with the first head. The details of this construction procedure are as follows:

The state set of the machine $M'$ is defined as $Q_{M'} = Q_M \cup \{q' \mid q \in Q_M\}$. Each transition of $M$ that moves both heads at once is simulated by two transitions that move the heads one after another in $M'$. Formally, for all $q, s \in Q_M$,.
If \( x, y \in \Sigma_{\text{out}} \), if \( \delta_M(q, x, y) = (s, +1, +1) \), we set \( \delta_{M'}(q, x, y) = (s', +1, 0) \). Furthermore, for all \( s \in Q_M, x, y \in \Sigma_{\text{out}} \), we set \( \delta_{M'}(s', x, y) = (s, 0, +1) \).

If a transition of \( M \) is stationary, i.e., is of the form \( \delta_M(q, x, y) = (s, 0, 0) \), it is a member of either an infinite sequence representing a loop (of length at most \( |Q_M| \)) in which \( M \) scans the symbols \( x \) and \( y \) without changing the head positions, or a finite sequence ending with acceptance, rejection, or the moving of some head. In the infinite-loop case, we set the corresponding transition in \( M' \) to \( \delta_M(q, x, y) = (q_{\text{rej}}, +1, 0) \). In the finite-sequence case, the value of \( \delta_{M'}(q, x, y) \) will be set to \( (q_{\text{acc}}, +1, 0) \) or \( (q_{\text{rej}}, +1, 0) \) if the sequence is ending with acceptance or rejection, respectively, and to the value of the final transition in the sequence otherwise.

Any transition of \( M \) that moves a single head is inherited without modification by \( M' \).

It may be the case that the new machine built according to these specifications moves its second head first. This problem can be handled easily by just rearranging the transition function to effectively “swap” the names of the two heads. (Such a simple swap is possible, because the fact that both heads scan the left end-marker symbol at the beginning means that it is only the transition function, and not the particular input string, that determines which head moves first.)

Consider the computation history of \( M' \) running on an input string \( w \). Keeping in mind that exactly one head moves at every step, the computation history can be split into sub-histories \( H_1, H_2, H_3, H_4, \ldots \), where only the first head moves during the odd-numbered sub-histories, and the second head moves during the even-numbered ones. Let us call \( H_1H_3H_5\cdots \) (i.e. the concatenation of the odd-numbered sub-histories) the odd part of the history, and \( H_2H_4H_6\cdots \) the even part.

Note that, if one visualizes the odd part of the history, one sees the first head moving in real time. Furthermore, the state sequence traversed during these moves is easy to trace step by step employing knowledge of \( M' \)'s transition function, except at the “joints” between sub-histories, where the machine’s state and the position of the second head make “leaps” corresponding to (possibly long) sequences of moves made by the second head while the first head was pausing. A similar observation can be made for the even part. Intuitively, both parts of the history can be thought of as describing the execution of a real-time automaton that momentarily “blacks out” as it switches from any \( H_i \) to \( H_{i+2} \), finding the machine’s state and the other head’s position updated to new values when it wakes up. Our strategy for real-time verification will follow directly from this observation, and the certificate will supply the necessary information to deal with the blackouts.

We will construct a real-time, finite-state verifier \( V \) that uses a single random bit to verify the language \( A \). The certificate alphabet of \( V \) is \( Q_{M'} \times \Sigma_{\text{out}} \times q \), with each symbol corresponding to a tuple of two states and two input symbols (including end-markers) of \( M' \).
The certificate $c_w$ for a string $w \in A$ will be a concise description of the state and head position values required by the two probabilistic paths of $V$ that will be assigned (as will be described shortly) to trace the odd and even parts of the computation history of $M'$ on $w$ to recover from the blackouts mentioned above:

$$c_w = (s_1, z_1, s_2, z_2)(s_3, z_3, s_4, z_4)(s_5, z_5, s_6, z_6) \cdots$$

The sequence above is to be interpreted as follows: For each $i$, the certificate “claims” that $M'$ will be in state $s_i$ at the end of $H_i$. For odd $i$, it claims that the first head will be scanning the input symbol $z_i$ at the end of $H_i$. Finally, for even $i$, the certificate claims that the second head will be scanning the symbol $z_i$ at the end of $H_i$.

Given an input and a certificate, $V$ starts by tossing a coin to choose which head of $M'$ to trace. If the first head is chosen, $V$ initiates a simulation of $M'$ from $q_0$, using its knowledge that the second head is paused on the symbol $\triangleright$ to determine the next state to transition to at each step. $V$ traces the first head of $M'$ with its own real-time head until it reaches a point during the simulation where $M'$ pauses the first head. (Recall that pausing its own head is impossible for $V$.) At that point, $V$ performs the following two operations at once: It verifies that it has just transitioned out of the state $s_1$ and that its input head is indeed scanning the symbol $z_1$, consistently so with the claims of the first certificate symbol $(s_1, z_1, s_2, z_2)$ (rejecting immediately if it discovers an inconsistency). It also advances its certificate head, and continues its simulation from the “wake-up” state $s_2$ with the transition $\delta_M'(s_2, z_1, z_2)$, assuming that the second head is now paused on the symbol $z_2$ as claimed by the certificate.

The procedure to be followed by $V$ if it chooses the second head at the beginning is similar, with some minor differences: In this case, $V$ moves the certificate head at its first step, leaving the first certificate symbol $(s_1, z_1, s_2, z_2)$ behind. (It stores $s_2$ and $z_2$ in its memory for later use.) $V$ starts simulating $M'$ from the state $s_1$, trusting the certificate’s claim that $M'$’s first head is paused on the symbol $z_1$, and using its own real-time head to mimic $M'$’s second head. This simulation proceeds until the point where $M'$ pauses its second head. $V$ checks whether the certificate’s claims about $s_2$ and $z_2$ were indeed consistent with its current information about the simulation state and head reading, consumes the next certificate symbol, and proceeds simulation from the new “wake-up” setting as described above if it has not discovered a lie of the certificate.

Each probabilistic branch of $V$ accepts if and only if the simulation reaches the accept state of $M'$. All $w \in A$ are accepted by $V$ with probability 1 when coupled with a proper certificate $c_w$ describing the sub-history transitions correctly. Whenever $w \notin A$, a $c_w$ that describes the history faithfully will lead both branches of $V$ to rejection. Any dishonest certificate trying to divert a branch to acceptance by giving false wake-up values will be caught out by the other branch that has direct access to the relevant state and head information, so all nonmembers of $A$ will be rejected with probability at least $\frac{1}{2}$.  

---

3 If $\delta_M'(s_2, z_1, z_2)$ happens to be a transition that moves the second head, $V$ rejects.

4 If the first simulated step from state $s_1$ does not move the second head, $V$ rejects.
3.2 Real-time verification beyond 1DFA(2)

Consider the language $L_{IK} = \{ a^i b^j c^k \mid i = j \text{ or } i = k \text{ or } j = k \}$, which is in 1DFA(3), but not in 1DFA(2) [8]. A real-time, finite-state verifier using a single random bit can verify $L_{IK}$ by checking certificates of the form $\sigma x^l$, where $\sigma$ is a ternary symbol that indicates which two of the three “segments” of the input string are claimed to be of the same length $l$. Depending on the values of $\sigma$ and the random bit, the verifier decides which segment to attempt to match with the certificate postfix $x^l$, and accepts only if this match succeeds.

More generally, for any $k > 0$, there exists a language of the form

$$L_n = \{ y_1 \# y_2 \# \cdots \# y_{2^n} \mid y_i \in \{ a, b \}^* \text{ and } y_i = y_{2n+1-i}, \text{ for } 1 \leq i \leq n \}$$

which can be recognized by a 1dfa($k+1$), but not by any 1nfa($k$) [13]. Such a language $L_n$ can be verified by a real-time, constant-space machine using $\lceil \log(n+1) \rceil$ random bits to split into $n+1$ paths that would compare the relevant segments of a certificate of the form $y_1 \# y_2 \# \cdots \# y_n$ with the corresponding input segments. So we have $\text{VER}(\text{con-space, con-random-bits, rt-input}) \setminus \text{1NFA}(k) \neq \emptyset$ for all $k \geq 1$.

We now exhibit a language that is verifiable in real time by constant-randomness finite-state machines, but is unrecognizable by any deterministic multihead automaton.

**Theorem 3.** $\text{VER}(\text{con-space, con-random-bits, rt-input}) \setminus \bigcup_k \text{1DFA}(k) \neq \emptyset$.

**Proof.** We will construct a verifier $V$ for the language $L_{\text{nonpal}}$, which was noted to be outside $\bigcup_k \text{1DFA}(k)$ in Section 2.

Every string $w$ in $L_{\text{nonpal}}$ matches the pattern $x\sigma y\sigma' z$, where $x, y, z \in \{ 0, 1 \}^*$ and $\sigma, \sigma' \in \{ 0, 1 \}$, such that $|x| = |z|$ and $\sigma \neq \sigma'$. The correct certificate $c_w$ for such an input will encode the positions of the “unmatching” symbols $\sigma$ and $\sigma'$ as follows:

$$c_w = 0|x|10|y|$$

$V$ tosses a single coin at the beginning of the computation to probabilistically “branch” to one of two “deterministic verifiers” $V_0$ and $V_1$, each of which checks the certificate $0^i 10^j$ in a different way, as described below.

Note that, if $0^i 10^j$ is indeed a correct certificate for the input, claiming that the two unmatching symbols are at positions $i + 1$ and $i + j + 2$, then the input string must be exactly $i + 1$ symbols longer than this certificate. $V_0$ checks this by moving the certificate head only once for every two moves of the input head over the input string until it passes over the 1 in the certificate. At that point, it switches to moving the certificate head at every step as well. If the certificate is of the correct length, the two heads will consume their right end-markers simultaneously, in which case $V_0$ will accept.

The task of $V_1$ is to assume that the certificate is well-formed in the sense described above, and accept if the two symbols at positions $i + 1$ and $i + j + 2$ really are unequal. This can be done by moving the certificate head at the same
speed as the input head, recording the symbol at position \((i + 1)\) in memory, and comparing it with the input symbol scanned at the step where the certificate string has been consumed completely.

If the input is a member of \(L_{\text{nonpal}}\), both \(V_0\) and \(V_1\) accept with the correct certificate. Otherwise, the input is a palindrome, and the certificate will either be malformed (and therefore be rejected by \(V_0\)), or the two symbols it points out will be equal, in which case it will be rejected by \(V_1\).

\(\square\)

4 Concluding remarks

Figure 1 summarizes the landscape of complexity classes covered in this paper. The \(\ominus\) symbol denotes that the two related sets are neither disjoint, nor a subset of one another.

We conjecture that

\[
\text{VER}(\text{con-space,con-random-bits,1way-input}) \subset \text{VER}(\text{con-space,con-random-bits,rt-input}),
\]

that is, restricting the input head to move in real-time yields machines which are not capable of verifying some languages that can be handled by verifiers with one-way input. The reasoning behind this conjecture is based on considerations of the following languages:

\[
L_{\text{match}} = \left\{ x\#y_1\#y_2\#\cdots\#y_k \mid x, y_i \in \{0, 1\}^+ \text{ for all } i, k > 0, \quad \text{and } y_i = x \text{ for some } i \right\}
\]

\[
L_2^+ = \left\{ w^* \mid w \in \{0, 1\}^* \right\}
\]

\[
L_2^* = \left\{ xwx \mid x, w \in \{0, 1\}^* \text{ and } |x| = |w| \right\}
\]

These languages, which are in \(\text{VER}(\text{con-space,con-random-bits,1way-input})\), seem to be beyond the capabilities of real-time verifiers. Verification of membership in these languages requires two input substrings (whose lengths are not bounded,
and which cannot therefore fit in a fixed amount of memory) to be “matched” in a certain sense. Furthermore, the start position of the second substring cannot be determined in a one-way pass without external help. Therefore, membership certificates have to contain information about both the position of the second substring and the content of these substrings. We suspect that it is impossible to design certificates from which real-time input machines can acquire these two pieces of information without getting tricked into accepting some illegal inputs.

For further study, it would be interesting to examine the power of real-time finite-state verifiers with less severe bounds on the amount of randomness that can be used, as well as real-time verification of debates between two opposing “provers” by similarly restricted machines. Restricting the verifiers further by imposing other conditions like reversibility is another possible direction.

Acknowledgments

The authors thank Martin Kutrib for his helpful answers to several questions. This work is supported by the Turkish Directorate of Strategy and Budget under the TAM Project number 2007K12-873.

References

1. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, USA (2009)
2. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. J. ACM 45(3), 501–555 (May 1998)
3. Arora, S., Safra, S.: Probabilistic checking of proofs: A new characterization of NP. J. ACM 45(1), 70–122 (Jan 1998)
4. Condon, A., Ladner, R.: Interactive proof systems with polynomially bounded strategies. Journal of Computer and System Sciences 50(3), 506–518 (1995)
5. Demirci, H.G., Say, A.C.C., Yakaryılmaz, A.: The complexity of debate checking. Theory of Computing Systems 57(1), 36–80 (Jul 2015)
6. Gezer, M.U., Say, A.C.C.: Constant-space, constant-randomness verifiers with arbitrarily small error. Information and Computation (in press)
7. Holzer, M., Kutrib, M., Malcher, A.: Complexity of multi-head finite automata: Origins and directions. Theoretical Computer Science 412(1-2), 83–96 (Jan 2011)
8. Ibarra, O.H., Kim, C.E.: On 3-head versus 2-head finite automata. Acta Informatica 4(2), 193–200 (1975)
9. Kutrib, M., Malcher, A., Wendlandt, M.: Set automata. International Journal of Foundations of Computer Science 27(02), 187–214 (2016)
10. Pin, J.E.: On reversible automata. In: Simon, I. (ed.) Proceedings of the first LATIN conference. pp. 401–416. Lecture Notes in Computer Science 583, Springer, São-Paulo, Brazil (1992)
11. Rosenberg, A.L.: On multi-head finite automata. IBM Journal of Research and Development 10(5), 388–394 (1966)
12. Say, A.C.C., Yakaryılmaz, A.: Finite state verifiers with constant randomness. Logical Methods in Computer Science 10(3) (Aug 2014)
13. Yao, A.C., Rivest, R.L.: k + 1 heads are better than k. J. ACM 25(2), 337–340 (Apr 1978)