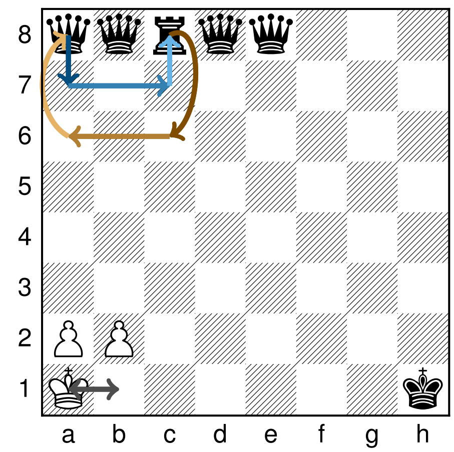
The Illusion of State in State-Space Models

William Merrill [1] **Jackson Petty** [1] **Ashish Sabharwal** [2]

Abstract

State-space models (SSMs) have emerged as a potential alternative architecture for building large language models (LLMs) compared to the previously ubiquitous transformer architecture. One theoretical weakness of transformers is that they cannot express certain kinds of sequential computation and state tracking (Merrill & Sabharwal, 2023a), which SSMs are explicitly designed to address via their close architectural similarity to recurrent neural networks (RNNs). *But do SSMs truly have an advantage (over transformers) in expressive power for state tracking?* Surprisingly, the answer is no. Our analysis reveals that the expressive power of SSMs is limited very similarly to transformers: SSMs cannot express computation outside the complexity class TC^0. In particular, this means they cannot solve simple state-tracking problems like permutation composition. It follows that SSMs are provably unable to accurately track chess moves with certain notation, evaluate code, or track entities in a long narrative. To supplement our formal analysis, we report experiments showing that Mamba-style SSMs indeed struggle with state tracking. Thus, despite its recurrent formulation, the "state" in an SSM is an illusion: SSMs have similar expressiveness limitations to non-recurrent models like transformers, which may fundamentally limit their ability to solve real-world state-tracking problems.




```
x = [0, 0, 1, 0, 0]
x[1], x[3] = x[3], x[1] # Swap 1, 3
```

Alice, Bob, Carl, Dan, and Emma each have a coin. All are dimes except Carl's. Alice and Carl trade coins.

Figure 1: We prove that SSMs, like transformers, cannot solve inherently sequential problems like permutation composition (S_5), which lies at the heart of state-tracking problems like tracking chess moves in source-target notation (see Section 3.2), evaluating Python code, or entity tracking. Thus, SSMs cannot, in general, solve these problems either.

1. Introduction

Recent theoretical work (Merrill & Sabharwal, 2023a) has shown that models built upon the transformer architecture are incapable of expressing inherently sequential computation. These results reveal a surprising limitation of transformers: they cannot express simple kinds of *state tracking* problems, such as composing sequences of permutations, which even simple recurrent neural networks (RNNs) can naturally express. In a different line of work, state space model (SSM) architectures (Gu et al., 2021; 2022a; Fu et al., 2023; Gu & Dao, 2023; Wang et al., 2024) have been introduced as an alternative to transformers, with the goal of achieving similar expressive power to RNNs for handling problems that are naturally stateful and sequential (Gu et al., 2021; 2022b). *But does the seemingly stateful design of SSMs truly enable them to solve sequential and state-tracking problems that transformers cannot?* If so, this would be a promising property of SSMs because state tracking is at the heart of large language model (LLM) capabilities such as tracking entities in a narrative (Heim, 1983; Kim & Schuster, 2023), playing chess under certain

[1]New York University [2]Allen Institute for AI. Correspondence to: William Merrill <willm@nyu.edu>, Jackson Petty <petty@nyu.edu>, Ashish Sabharwal <ashishs@allenai.org>.

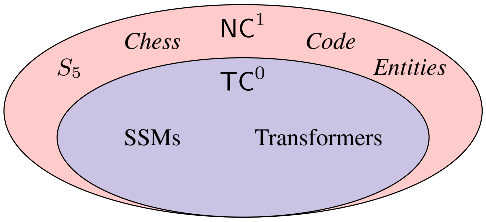


Figure 2: Complexity hierarchy within NC^1. Transformers can only recognize languages within TC^0 (Merrill & Sabharwal, 2023a), and we show the same for SSMs (Theorems 4.1 and 4.2). Thus, both architectures cannot express the "hard state tracking" captured by NC^1-complete problems like S_5, which *can* be straightforwardly expressed by RNNs. The figure assumes the widely held conjecture $\mathsf{TC}^0 \neq \mathsf{NC}^1$.

notation[1], or evaluating code. This would motivate further research into SSM architectures and their deployment as the next generation of LLMs.

In this work, we show that the apparent stateful design of SSMs is an *illusion* as far as their expressive power is concerned. In contrast to the suggestion by Gu et al. (2021; 2022b) (and, perhaps, a broader belief in the community) that SSMs have expressive power for state tracking similar to RNNs, we prove theoretically that linear and Mamba-style SSMs, like transformers, cannot express inherently sequential problems, including state-tracking problems like composing permutations that RNNs can easily express. Further, our experiments confirm our formal predictions: both transformers and these SSMs cannot learn to compose permutations with a fixed number of layers, whereas RNNs can compose permutations with just a single layer. Our results imply that arguments that SSMs have an advantage over transformers due to being "more recurrent" or capable of tracking state are misguided. In fact, the SSM architectures we consider are just as theoretically unequipped for state tracking and recurrent computation as transformers are.

We first establish the theoretical weakness of linear SSMs and near generalizations by proving they are in the complexity class L-uniform TC^0, which has been previously shown for transformers (Merrill & Sabharwal, 2023a). This implies these SSMs cannot solve inherently sequential problems (formally, problems that are NC^1-hard), including state-tracking problems like permutation composition (Liu et al., 2023). Permutation composition is a fundamental problem at the heart of many real-world state-tracking problems such as playing chess, evaluating code, or tracking entities in a narrative (Figure 1), implying solutions to these problems, too, cannot be expressed by SSMs, at least in the worst case.

At first glance, our results may appear to contradict Gu et al. (2021)'s claim that linear SSMs can simulate general recurrent models, which can express permutation composition. But the contradiction is resolved by a difference in assumptions: Gu et al. (2021) relied on *infinite depth* (number of layers) to show that SSMs could simulate RNNs. We, on the other hand, analyze the realistic setting with a bounded number of layers, under which we find that SSMs cannot simulate the recurrent state of an RNN and, in fact, suffer from similar limitations as transformers for state tracking.

Our empirical investigation shows that, in practice, both SSMs with the Mamba architecture (Gu & Dao, 2023) and transformers do *not* learn to solve the permutation composition state-tracking problem with a fixed number of layers, while simple RNNs can do so with just one layer. This provides empirical support for our theoretical separation in expressive power for state tracking between SSMs and true recurrent models. We also find that both transformers and SSMs struggle compared to RNNs on state-tracking problems less complex than permutation composition where it is not known whether they can express a solution. Thus, in practice, SSMs may struggle not just on the hardest state-tracking problems like permutation composition but also on easier variants.

Finally, we propose two minimal extensions of linear SSMs that increase their expressive power for state tracking, allowing them to solve permutation composition. These extensions, however, may come with a cost: they may negatively impact parallelism as well as learning dynamics. We view it as an interesting open question whether it is possible to develop SSM-like models with greater expressivity for state tracking that also have strong parallelizability and learning dynamics, or whether these different goals are fundamentally at odds, as Merrill & Sabharwal (2023a) suggest.

2. Background

We first present the SSM architectures we will analyze (Section 2.1). Our analysis of the state tracking capabilities of SSMs borrows deeply from the circuit complexity and algebraic formal language theory literature. We thus review how circuit complexity can be used to analyze the power of neural networks (Section 2.3) and how state-tracking problems can be captured algebraically and analyzed within the circuit complexity framework (Section 3.1).

2.1. Architecture of State-Space Models

SSMs are a neural network architecture for processing sequences similar in design to RNNs or linear dynamical systems. SSMs have been suggested to have two potential advantages compared to transformers owing to their recurrent formulation: faster inference and, possibly, the ability to better express inherently sequential or stateful problems (Gu et al., 2021; 2022b). Several architectural variants of

[1] The hardness of chess state tracking holds with (source, target) notation, but standard notation may make state tracking easier.

SSMs have been proposed, including S4 (Gu et al., 2022a) and Mamba (Gu & Dao, 2023). Recently, SSMs have been shown to achieve strong empirical performance compared to transformers in certain settings, particularly those involving a long context (Gu & Dao, 2023; Wang et al., 2024).

SSMs consist of *state-space layers*, which can be thought of as simplified RNN layers. We consider two variants of the state-space layer: the linear SSM layer (of which S4 is a special case; Gu et al., 2022a) and the extended S6 layer used by Mamba (Gu & Dao, 2023).

Definition 2.1 (Linear SSM layer; e.g. S4)**.** Given a sequence of embeddings or previous states $x_1, \dots, x_n \in \mathbb{R}^m$, the *recurrent form* of a linear SSM layer defines a new sequence of states $h_1, \dots, h_n \in \mathbb{R}^d$ using learned parameter matrices $A \in \mathbb{R}^{d \times d}$ and $B \in \mathbb{R}^{d \times m}$. For each $1 \leq i \leq n$,

$$h_i = Ah_{i-1} + Bx_i. \tag{1}$$

The *convolutional form* of the linear SSM layer defines the same[2] $h_1, \dots, h_n$ but is unrolled as a summation of terms:

$$h_i = \sum_{j=1}^{i} A^{i-j} B x_j. \tag{2}$$

The layer outputs $y_i = Ch_i + Dx_i$ where $C \in \mathbb{R}^{d \times m}$ and $D \in \mathbb{R}^{m \times m}$.

S4 chooses a specific parameterization of a linear layer. A full S4 model is a cascade of such layers and feedforward layers, analogous to how transformers alternate multihead-self-attention layers with feedforward layers.

The S6 layer used by Mamba (Gu & Dao, 2023) generalizes a linear SSM layer by adding a *selection mechanism* inspired by the dynamic gating in LSTMs (Hochreiter & Schmidhuber, 1997) and GRUs (Cho et al., 2014).

Definition 2.2 (S6 layer)**.** An S6 layer is parameterized by a *diagonal* matrix $A \in \mathbb{R}^{d \times d}$, a vector $\delta \in \mathbb{R}^d$, and affine projections $s_\delta : \mathbb{R}^m \to \mathbb{R}^d, s_B : \mathbb{R}^m \to \mathbb{R}^d$, and $s_C : \mathbb{R}^d \to \mathbb{R}^m$. Let τ be softplus. The S6 layer is:

$$\begin{aligned} h_i &= \exp(\delta_i A) h_{i-1} + B_i x_i, \\ \text{where } \delta_i &= \tau(\delta + s_\delta(x)) \text{ and} \\ B_i &= s_B(x). \end{aligned} \tag{3}$$

This implies the convolutional form:

$$h_i = \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i} \exp(\delta_k A) \right) B_j x_j. \tag{4}$$

[2]The two forms express the same function over $\mathbb{R}$ or any other distributive datatype. Over floating points (Appendix A), they are not guaranteed to be the same, but we must assume the error is negligible for them to be well-defined and usable in practice.

Finally, to compute y_i, C is made input dependent C_i and computed via a projection in the same manner as B_i. The layer output is then $y_i = C_i h_i + x_i$.

In practice, a crucial detail for training SSMs is the initialization of the weight matrices. Our main results (Theorems 4.1 and 4.2) will apply for any linear SSM (including S4) as well as S6, independent of the specific values of its weights. In contrast to S4 and S6, H3 (Fu et al., 2023) is not a true SSM because the context is not represented by a single vector. Rather, its architecture resembles a transformer with SSM components. Analyzing H3 is beyond our focus, but we believe our ideas could be extended to H3 in future work.

2.2. Numeric Datatype

Circuit-complexity analysis of neural networks depends to some degree on low-level details about arithmetic and the underlying datatype $\mathbb{D}$ used in the network's computation graph. We can think of $\mathbb{D}$ as parameterized by the number of bits available to represent a number in $\mathbb{D}$. For instance, non-negative integers in $[0, 2^p]$ use p bits, signed integers in $[-2^p, 2^p]$ use $p + 1$ bits, FP16 uses 16 bits, etc.

Our main results (Theorems 4.1 and 4.2) will go through for any datatype $\mathbb{D}$ for which the following 3 operations are efficiently parallel-computable, i.e., are in the complexity class L-uniform TC^0 (to be defined shortly in Section 2.3):

1. Iterated addition, i.e., summing n numbers in $\mathbb{D}$
2. Iterated product, i.e., multiplying n numbers in $\mathbb{D}$
3. Matrix powering, i.e., computing the n-th power of a fixed-size $k \times k$ matrix over $\mathbb{D}$

When $\mathbb{D}$ is any finite-precision datatype, i.e., has a fixed number of bits available (e.g., 16 or 64), then these operations are easily seen to be in L-uniform TC^0. As Merrill & Sabharwal (2023b) argue, however, finite-precision datatypes severely limit the expressivity of neural architectures from a formal perspective (e.g., finite-precision transformers cannot represent uniform attention), motivating the use of parameterized datatypes that can (approximately) represent any number with a sufficiently large parameter. Interestingly, when $\mathbb{D}$ is the datatype of n-bit integers, all of the above operations are known to be in L-uniform TC^0 (Hesse, 2001; Mereghetti & Palano, 2000). Realistically, however, neural model implementations use floating point numbers with much fewer than n bits.

Concretely, we imagine working with the **log-precision floating point** model used by Merrill & Sabharwal (2023b) to analyze transformers, which we will show satisfy all these properties. For some fixed constant $c \in \mathbb{Z}^+$, a $c \log n$ precision float is a tuple $\langle m, e \rangle$ where m, e are signed integers together taking $c \log n$ bits. Using $|x|$ to mean the number of bits used to represent integer x, this float represents the value $m \cdot 2^{e-|m|+1}$.

Unlike for integers, arithmetic operations over log-precision floats are not closed. That is, the product $\phi_1 \times \phi_2$ of two p-precision floats is a well-defined number but may not be exactly representable as a p-precision float. It is thus necessary to define approximate versions of these operations when formalizing log-precision floating-point arithmetic. To this end, Merrill & Sabharwal (2023a) define a natural notion of approximate iterated addition over log-precision floats and show that it is computable in L-uniform TC^0. Following Merrill & Sabharwal (2023a)'s definition of iterated addition for floats, we define iterated multiplication and matrix powering over any datatype $\mathbb{D}$ as the result of treating the numbers as reals, performing exact arithmetic, and casting the exact output ϕ back to $\mathbb{D}$, denoted $\mathsf{cast}_{\mathbb{D}}(\phi)$.

In Appendix A, we extend the arguments of Hesse (2001) and Mereghetti & Palano (2000) for integers to show that iterated product and matrix powering over log-precision floats are also computable in L-uniform TC^0 (cf. Lemmas 2.3 and 2.4 in Appendix A).

Lemma 2.3 (Iterated float product)**.** *Let $\phi_1, \ldots, \phi_z$ be $c \log n$ precision floats and $z \leq n$. Then the iterated float product $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0.*

Lemma 2.4 (Float matrix power)**.** *Let $k, c \in \mathbb{Z}^+$ be fixed constants. Let M be a $k \times k$ matrix over $c \log n$ precision floats. Let $z \leq n, z \in \mathbb{Z}^+$. Then float matrix power M^z can be computed in L-uniform TC^0.*

Combined with the result for iterated addition from Merrill & Sabharwal (2023a), this establishes that the three needed properties are met for log-precision floats.

2.3. Limits of Transformers via Circuit Complexity

A line of recent work has used circuit complexity and logic formalisms to identify the expressiveness limitations of transformers on reasoning problems (Angluin et al., 2023; Merrill & Sabharwal, 2023a; Liu et al., 2023; Chiang et al., 2023; Merrill & Sabharwal, 2023b; Hao et al., 2022); see Strobl et al., 2023 for a survey. In particular, Merrill & Sabharwal (2023a) showed that transformers can only solve problems in the complexity class TC^0, which is defined as the set of problems that can be recognized by constant-depth, polynomial-size threshold circuit families. Such circuits, in addition to having standard AND, OR, and NOT gates (of arbitrary fan-in), can also use threshold gates that output 1 iff at least k of the inputs are 1, where k is a parameter of the gate. Informally, TC^0 can be thought of as the class of problems that can be solved with extremely parallelized (constant-depth) computation.[3]

Problems outside TC^0, corresponding to problems that are inherently sequential and thus cannot be parallelized, cannot be solved by transformers. No problems in polynomial time are known unconditionally to be outside TC^0, but unless the widely held conjecture that $\mathsf{TC}^0 \neq \mathsf{NC}^1$ is false, many simple NC^1-hard problems are outside TC^0. In particular, this includes simulating finite automata (NC^1-complete), evaluating boolean formulas (NC^1-complete), determining graph connectivity (L-complete), and solving linear equations (P-complete). These problems have already been shown to be inexpressible by transformers (Merrill & Sabharwal, 2023a). By showing that SSMs can be simulated in TC^0, we will establish that they also cannot be solved by SSMs.

3. State Tracking

Informally, a state-tracking problem is a problem where the text specifies some sequence of updates to the state of the world, and the goal of the problem is to say what the resulting world state is after the updates have been applied in sequence. This circuit complexity view on the power of neural networks can be combined with other insights from algebraic formal language theory to analyze the kinds of state tracking that SSMs can express. In particular, this theory comprehensively shows us which kinds of state-tracking problems are (likely) not in TC^0. This will, in turn, allow us to find examples of hard state tracking that models like SSMs will not be able to solve.

3.1. State Tracking as a Monoid Word Problem

From the perspective of algebraic formal language theory, state tracking over a finite world can be captured as a *word problem* on a *finite monoid* (Liu et al., 2023).[4] Different updates to the world become different elements in the monoid, and resolving the final world state after all the updates have been applied is equivalent to computing the product of a sequence of elements (also called a "word").

Definition 3.1 (Word Problem)**.** Let M be a finite set, and $(M, \cdot)$ a finite monoid (i.e., M with identity and associative multiplication). The word problem for M is to reduce sequences in M^* under multiplication; that is, send $m_0 m_1 \cdots m_k$ to $m_0 \cdot m_1 \cdot \ldots \cdot m_k \in M$. Solving the word problem requires reducing sequences of arbitrary length.

Example 3.2. Consider the monoid $\{0, 1\}$ where $\cdot$ is addition modulo 2. The word problem involves computing the parity of a string, e.g., $0011 \mapsto 0$. From a state-tracking perspective, this monoid captures a world with a single light

[3] We use TC^0 to mean L-uniform TC^0, meaning the circuit family is constructible by a Turing machine that runs in space logarithmic in the size of the input (cf. Merrill & Sabharwal, 2023a; Strobl et al., 2023). We believe our results could be extended from L-uniform TC^0 to $\mathsf{DLOGTIME}$-uniform TC^0 using techniques similar to Merrill & Sabharwal (2023b) for composing TC^0 circuits in a way that preserves $\mathsf{DLOGTIME}$ uniformity.

[4] We consider only finite monoids for simplicity, but, in principle, it would be possible to extend this approach to infinite (e.g., finitely generated) monoids as well.

switch. The identity 0 corresponds to no action whereas 1 is an update that flips the switch.

Modeling state tracking with word problems lets us draw connections between circuit complexity and abstract algebra to understand which word problems are "hard" to solve. Krohn & Rhodes (1965) established that not all word problems are created equal: some, like Example 3.2, are in TC^0, while others are NC^1-complete, requiring recurrent processing to solve (Immerman & Landau, 1989; Barrington, 1989). Because we will show SSMs can be simulated in TC^0, it follows that NC^1-complete state-tracking problems cannot be solved by SSMs (cf. Figure 2).

Whether or not a word problem is NC^1-complete depends on the algebraic structure of the underlying monoid.[5] Barrington (1989) showed that the word problem of every finite non-solvable[6] group is NC^1-complete. That non-solvable groups have NC^1-complete word problems is notable because of the ubiquity with which non-solvable groups show up in tasks involving state tracking. The canonical example of an NC^1-complete word problem is that of S_5, the symmetric group on five elements that encodes the permutations over five objects. As an immediate instantiation of this, consider a document describing arbitrarily many sequences of transpositions: *"swap ball 1 and 3, swap ball 3 and 5, swap ball 4 and 2, ..."*[7]. Being able to answer the question *"where does ball 5 end up?"* requires solving the S_5 word problem. Beyond permutations, Figure 1 shows how many natural state-tracking problems like tracking chess moves, evaluating code, or tracking entities also encode the structure of S_5, meaning these state-tracking problems also cannot be expressed by a model in TC^0. Rather, in order to solve these problems, the depth of the model would have to be expanded to accommodate longer inputs.

Although the S_5 word problem is canonical, in this paper we will consider the word problem on a closely related group A_5: the *alternating* group on five elements. We do this for simplicity: A_5 is a subgroup of S_5 containing only even permutations, and is the smallest non-solvable subgroup. We will compare the word problem on A_5 to two other baseline groups: $A_4 \times \mathbb{Z}_5$, a non-abelian but solvable group; and $\mathbb{Z}_{60}$, an abelian group encoding mod-60 addition. We choose these groups as points of comparison because they all have 60 distinct elements, meaning that the difficulty in learning their word problems will come only from the complexity of learning the group multiplication operation.

[5] We focus on word problems on *groups*, which are monoids with inverses.

[6] Formally, a group G is solvable exactly when there is a series of subgroups $1 = G_0 < G_1 < \cdots < G_k = G$ such that G_{i-1} is normal in G_i and G_i/G_{i-1} is abelian.

[7] Without less of generality, any permutation can be factored into a sequence of transpositions, or swaps. This means the transpositions over five elements are a generator for S_5.

3.2. Encoding S_5 in Chess State Tracking

Figure 1 already gives some intuition into how state-tracking problems encode S_5. Out of these examples, the most intricated case is chess. We now give a proper reduction from S_5 to tracking chess moves, showing formally that not just S_5, but chess state tracking as well, is NC^1-complete.

We define the chess state-tracking problem as follows:

- **Input:** The **state of a chessboard** as well as a **sequence of chess moves**, where each move is specified as a tuple (source square, target square). Note that this differs from the standard notation that represents a move as a piece along with target square and potential disambiguating information.
- **Output:** The resulting board state after starting in the initial board state and applying the sequence of moves one after another, ignoring draws. If any move is illegal given its intermediate board state, we enter a special null board state.

We show that S_5 can be reduced to chess state tracking, establishing the NC^1-completeness of chess state tracking. In other words, we can map any S_5 sequence to a sequence of chess moves and read off the answer to the S_5 instance from the final chessboard state.

Proposition 3.3. *S_5 can be reduced to chess state tracking via NC^0 reductions.*

Proof. The idea, as illustrated in Figure 1, is to map each S_5 element to a sequence of chess moves that permutes five pieces on the chessboard. Then, the final chessboard state will allow us to determine the composition of the permutation sequence. We defer a detailed proof to Appendix B. □

Since S_5 is NC^1-complete under AC^0 reductions:

Corollary 3.4. *The chess state-tracking problem is NC^1-complete under AC^0 reductions.*

Similar reductions can be constructed for evaluating Python or tracking entities in a dialog, as suggested in Figure 1. For another example of a similar reduction used to prove NC^1-completeness, we refer the reader to Theorem 3.2 of Feng et al. (2023).

4. SSMs Can be Simulated in TC^0

In this section, we show that the convolutional form of an SSM can be simulated in TC^0. Assuming the convolutional form of the model computes the same function as the recurrent form, this implies that SSMs, in whatever parameterization, cannot solve inherently sequential problems, despite their appearance of recurrence and statefulness. We first show containment in TC^0 for the simple non-gated variant of generalized S4 models (Theorem 4.1), and then show that the proof goes through for generalized S6 models as well (Theorem 4.2). The main idea in both proofs is that matrix powering can be computed in TC^0 (Mereghetti & Palano, 2000), and computing the convolutional form of an SSM can essentially be reduced to matrix powering.

4.1. Simulating Linear SSMs with a TC^0 Circuit Family

Theorem 4.1. *For any log-precision SSM M with the S4 architecture, there exists an* L*-uniform* TC^0 *circuit family that computes the same function as M's convolutional form.*

Proof. It suffices to construct an L-uniform TC^0 circuit family to simulate a single layer. Then, the full SSM can be simulate by generating this circuit multiple times, routing the output bits from one layer as the inputs to the next. This can be done with log-space overhead by simply storing a counter that tracks the index of the current input gates.

Recall that the S4 convolutional form Eqn. (2) is $h_i = \sum_{j=1}^{i} A^{i-j} B x_j$. Crucially, we use the fact that matrix powering over floats is in L-uniform TC^0 (Lemma 2.4, extending Mereghetti & Palano, 2000) to print, for each i, j, a TC^0 circuit family that computes $\pi_{ij} \triangleq A^{i-j}$. Next, we use the fact that fixed-arity arithmetic over floats is in L-uniform TC^0 to print, for each i, j, a circuit that computes

$$\tau_{ij} \triangleq \pi_{ij} B x_j = A^{i-j} B x_j.$$

Since iterated addition over floats is in L-uniform TC^0 (Merrill & Sabharwal, 2023a, extending Hesse, 2001; Chiu et al., 2001 for integers), we can compute $h_i \triangleq \sum_{j=1}^{i} \tau_{ij}$. Finally, we compute y_i from x_i and h_i in TC^0 via simple arithmetic operations, completing the proof. □

4.2. Simulating S6 with a TC^0 Circuit Family

Theorem 4.2. *For any log-precision SSM M with the S6 architecture, there exists an* L*-uniform* TC^0 *circuit family that computes the same function as M's convolutional form.*

Proof. The convolutional form of the S6 layer is given in Eqn. (4). For each i we print an L-uniform TC^0 circuit computing δ_i, B_i and C_i. Since A is diagonal, iterated matrix multiplication is reducible to iterated scalar multiplication, which is in L-uniform TC^0 (Lemma 2.4). so we can compute the following in TC^0:

$$\pi_{ij} \triangleq \prod_{k=j+1}^{i} \exp(\delta_k A).$$

We then conclude analogously to Theorem 4.1, first computing τ_{ij} from π_{ij} and B_i, then computing $h_i = \sum_{j=1}^{i} \tau_{ij}$, and finally computing y_i from x_i, h_i, and C_i in TC^0. □

4.3. Discussion

Theorems 4.1 and 4.2 establish that SSMs, like transformers, can only express solutions to problems in the class TC^0. This means that SSMs cannot solve NC^1-hard problems like evaluating boolean formulas or graph connectivity. In particular, it shows that they are limited as far as their state tracking capabilities as they are unable to compose permutations (solve the S_5 word problem):

Corollary 4.3. *Assuming $\mathsf{TC}^0 \neq \mathsf{NC}^1$, no log-precision SSM with the S4 or S6 architecture can solve the word problem for S_5 or any other NC^1-hard problem.*

In contrast, RNNs can easily express S_5 via standard constructions that encode finite-state transitions into an RNN (Minsky, 1954; Merrill, 2019). This shows that SSMs cannot express some kinds of state tracking and recurrence that RNNs can. This tempers the claim from Gu et al. (2021, Lemma 3.2) that SSMs have the expressive power to simulate RNNs, which relied on the assumption that SSMs can have *infinite depth*. In a more realistic setting with a bounded number of layers, our results show SSMs cannot express many state-tracking problems, including those which can be solved by fixed-depth RNNs.

5. Extending the Expressive Power of SSMs

We have shown that S4 and S6, despite their seemingly "stateful" design, cannot express problems outside TC^0, which includes state tracking like S_5. We show how SSMs can be extended to close the gap in expressive power with RNNs, allowing them to express S_5. Two simple extensions can bring about this increase in expressive power: adding a nonlinearity to make the SSM more like an RNN or allowing the A matrix to be input-dependent to make the SSM more like a weighted finite automaton (WFA; Mohri, 2009).

5.1. Via Nonlinearities

Concretely, an *RNN-SSM layer* with a step activation function can be defined as follows:

$$h_i = \operatorname{sgn}\left(A h_{i-1} + B x_i\right). \tag{5}$$

After this change, the SSM no longer has a straightforward convolutional form. However, its recurrent form is effectively an RNN, and can therefore solve S_5:

Theorem 5.1. *For any regular language L (including the word problem for S_5), there exists a one-layer log-precision RNN-SSM that recognizes L.*

Proof. The standard constructions for simulating automata with RNNs (cf. Minsky, 1954; Merrill, 2019) apply. □

Note that adding nonlinearities to the output of an SSM layer (as in Mamba) is not the same thing as an RNN-SSM.

Rather, an RNN-SSM has nonlinearities applied after each recurrent update to the state.

5.2. Via Input-Dependent Transition Matrices

Another completely different way to get greater expressive power is to let A matrix to be input-dependent. To illustate this, we define and analyze the *WFA-SSM layer*. Let $A(x_i) = I + s_A(x_i)$. The recurrent form becomes:

$$h_i = A(x_i)h_{i-1} + Bx_i. \tag{6}$$

This means the convolutional form computes an *iterated product* of a sequence of matrices rather than powering a matrix as for S4 and S6 (cf. Section 2.1):

$$h_i = \sum_{j=1}^{i} \left(\prod_{k=j+1}^{i} A(x_k) \right) Bx_j. \tag{7}$$

Unlike matrix powers, iterated matrix products cannot, in general, be computed in TC^0 (Mereghetti & Palano, 2000). This means that the argument from Theorem 4.1 will not go through for WFA-SSMs. In fact, we can show that WFA-SSMs gains expressive power beyond TC^0:

Theorem 5.2. *For any regular language L over vocabulary Σ (including the word problem for S_5), there exists a one-layer log-precision WFA-SSM that recognizes $\$L$, where $\$ \notin \Sigma$ is a special beginning-of-string symbol.*

Proof. It suffices to show that an WFA-SSM can simulate a deterministic finite automaton (DFA). We do this via a transition monoid construction. For any $w \in \Sigma^*$, let $\delta_w : Q \to Q$ be the function mapping a state to its eventual destination state after w is read from that state. For any DFA, this set of functions forms a finite monoid (the *transition monoid*) under composition, following from the Myhill-Nerode theorem (Hopcroft et al., 2001). Further, each monoid element δ_w can be represented as a boolean transition matrix, making matrix multiplication isomorphic to monoid composition.

Computing the transition monoid of a DFA allows recognizing valid words: compute the monoid element for a word by multiplying the elements for its tokens and then check whether the initial state maps to an accepting state. In fact, a standard way to solve monoid word problems (e.g., for S_5) with a DFA is simply to construct a DFA whose transition monoid is the monoid of interest.

Fix a DFA and its transition monoid δ. To complete the proof, we show that there exists an SSM that, for all $w \in \Sigma^*$, computes δ_w given input $x = \$w$. We view indices in h_i as states and define $B\$$ as 1 at each accepting state, and 0 elsewhere. For all other σ, we let $B\sigma = \vec{0}$. This reduces the convolutional form of h_i to have a single term:

$$h_i = \prod_{k=2}^{i} A(x_k).$$

Now, let $A(\sigma) = \delta_\sigma$ for $\sigma \in \Sigma$. It follows that

$$h_i = \prod_{k=2}^{i} A(x_k) = \prod_{k=2}^{i} \delta_{x_k}.$$

Since $x = \$w$, we conclude that $h_{|x|}$ is δ_w. $\square$

5.3. Discussion

Theorems 5.1 and 5.2 show that two minimal extensions of the SSM enable expressive power outside TC^0, allowing the model to solve hard state-tracking problems:

Corollary 5.3. *There exist a one-layer log-precision RNN-SSM and WFA-SSM that express the word problem for S_5 (with a beginning-of-string symbol), and these these SSMs cannot be simulated in TC^0.*

But would these variants of SSMs be feasible to use in practice? Besides expressive power, there are two competing practical concerns that might make these extensions problematic: parallelism and the impact on learning dynamics.

Parallelism. To be used effectively in an LLM, a model architecture must be parallelizable on practical hardware. Architectures in TC^0 are parallelizable by design (Merrill & Sabharwal, 2023a), but architectures in NC^1 may still be parallelizable to log depth even if they cannot be parallelized to constant depth. For the WFA-SSM, the bottleneck would be computing iterated matrix product with a log-depth computation graph. Similarly, for the RNN-SSM, the bottleneck would be computing the state updates with a log-depth binary tree rather than left to right. If this could be accomplished on practical hardware, these architectures could be parallelizable enough to scale on modern hardware.

Learning Dynamics. Another potential concern for these SSM variants compared to the original SSM is whether the learning dynamics are as good. In particular, for the WFA-SSM, an iterated product of matrices may lead to vanishing gradient issues. However, this is already potentially an issue for the S6 architecture, where the selective gating involves computing an iterated product of scalars.

6. Can SSMs Learn Permutations in Practice?

Having established theoretical limitations of SSMs for state tracking, we now empirically test how well SSMs can learn such tasks, focusing on the word problem for A_5. Since this problem is NC^1-complete and both transformers and SSMs

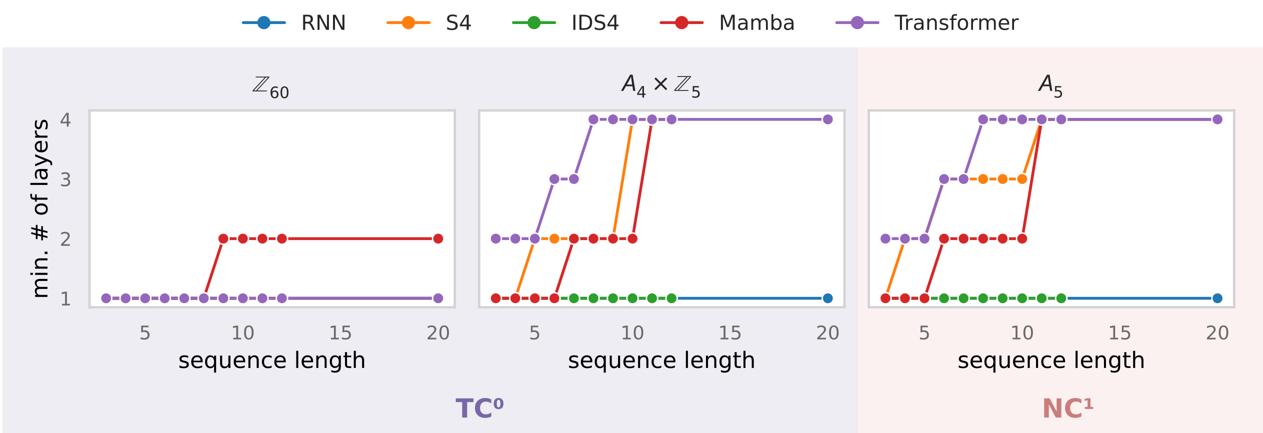


Figure 3: Minimum number of layers required to attain $> 90\%$ validation accuracy on group multiplication problems by sequence length and group. RNN and IDS4 models of constant depth can solve arbitrarily long sequences, while transformer, S4, and Mamba models require depths monotonically increasing in sequence length.

can only express functions in TC^0, to solve instances of this problem these models should require a dynamic depth that grows with the input length.

Setup. We model word problems (see Section 3.1) as a token-tagging task. Models are given as input a sequence of elements $g_0 g_1 g_2 \cdots g_n$ drawn from one of A_5, $A_4 \times \mathbb{Z}_5$, or $\mathbb{Z}_{60}$, and for each step t_i must predict the result of multiplying the first i elements of the sequence together. Modeling the problem as a tagging task rather than as a sequence classification task provides the models with more supervision during training, making it as easy as possible to learn the correct function. We tokenize inputs such that each element gets a unique token. We train models on sequences of length n for successively larger values of n and report full-sequence accuracy on a validation set.[8] To validate the predictions of SSMs and transformers being depth-bounded for expressing the word problem for A_5, we sweep over the number of layers each S4, Mamba and transformer model has, and compare these results to a single-layer simple RNN and our proposed single-layer Input-Dependent S4 (IDS4) architecture.

Results. Figure 3 shows that single-layer RNN and IDS4 models learn the word problem for arbitrarily long sequences from all three groups. In contrast, transformer, S4, and Mamba models require depth monotonically increasing in sequence length to attain good accuracy on a validation set on non-commutative groups. We draw three main conclusions from this:

1. Mamba and S4 show the same qualitative limitations as transformers on the inherently sequential task A_5: longer A_5 sequences require deeper models. This is consistent with both transformers, S4, and Mamba being in TC^0. On the other hand, RNNs (which resemble automata; Merrill, 2019) and IDS4 can solve NC^1-complete problems (McKenzie et al., 1991) including the A_5 word problem.
2. S4, Mamba, and transformers require greater depth even $A_4 \times \mathbb{Z}_5$, which can be theoretically expressed by TC^0 circuits. Although transformer and Mamba models of a given depth perform as good or better on $A_4 \times \mathbb{Z}_5$ as they on A_5, they still require increasingly many layers to handle proportionally longer sequences. There are two possible interpretations of this. First, it could be that while these word problems are expressible in TC^0, they cannot be expressed by S4, Mamba, or transformers (which can each likely recognize only a proper subset of TC^0). On the other hand, it is possible that these word problems *are* expressible by transformers, S4, and Mamba but that effectively learning a constant-depth solution is difficult.
3. Despite this limitation, S4 and Mamba appear *empirically better* at approximate state tracking on the non-commutative tasks than the transformer. For length-n sequences from $A_4 \times \mathbb{Z}_5$ or A_5, the transformer requires at least as many (and frequently fewer) layers as S4 or Mamba to solve the task.

7. Conclusion

We have shown that SSMs, like transformers, can only express computation in the complexity class L-uniform TC^0. This means they cannot solve inherently sequential problems like graph connectivity, boolean formula evaluation, and—

[8] We always include the full set of 3600 pairwise sequences of length 2 in the training data, along with the training split of length-n sequences.

of particular interest for state tracking—the permutation composition problem S_5. S_5 can be naturally expressed by true recurrent models like RNNs and captures the essence of hard state tracking due to its NC^1-completeness. In practice, one-layer RNNs can easily learn a task capturing S_5 SSMs require depth growing with the sequence length. These results reveal that SSMs cannot truly track state: rather, they can only solve simple state-tracking problems for which shallow shortcuts exist (Liu et al., 2023).

We also showed that simple extensions of SSMs can express S_5, although this comes with potential drawbacks as far as parallelism and learning dynamics. In future work, it would be interesting to more thoroughly explore the practical viability of our SSM extensions. Ultimately, this line of work has the potential to unlock new neural architectures that balance the parallelism of transformers and SSMs with full expressive power for state tracking, enabling LLMs that can benefit from scale while enjoying a greater capacity to reason about games, code, and language.

Broader Impact

This paper aims to advance the foundational understanding of state-space architectures for deep learning. Such work can affect the development and deployment of deep learning models in a variety of ways, which in turn can have societal impacts. However, we find it difficult to meaningfully speculate about or anticipate these downstream impacts here.

References

Angluin, D., Chiang, D., and Yang, A. Masked hard-attention transformers and Boolean RASP recognize exactly the star-free languages, 2023. arXiv:2310.13897.

Barrington, D. A. Bounded-width polynomial-size branching programs recognize exactly those languages in nc1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(89)90037-8. URL https://www.sciencedirect.com/science/article/pii/0022000089900378.

Chiang, D., Cholak, P., and Pillay, A. Tighter bounds on the expressivity of transformer encoders. In *ICML*, 2023.

Chiu, A., Davida, G. I., and Litow, B. E. Division in logspace-uniform NC1. *RAIRO Theor. Informatics Appl.*, 35:259–275, 2001.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In Moschitti, A., Pang, B., and Daelemans, W. (eds.), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://aclanthology.org/D14-1179.

Feng, G., Zhang, B., Gu, Y., Ye, H., He, D., and Wang, L. Towards revealing the mystery behind chain of thought: A theoretical perspective. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=qHrADgAdYu.

Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Re, C. Hungry hungry hippos: Towards language modeling with state space models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=COZDy0WYGg.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces, 2023.

Gu, A., Johnson, I., Goel, K., Saab, K. K., Dao, T., Rudra, A., and Re, C. Combining recurrent, convolutional, and continuous-time models with linear state space layers. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=yWd42CWN3c.

Gu, A., Goel, K., and Re, C. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022a. URL https://openreview.net/forum?id=uYLFoz1vlAC.

Gu, A., Goel, K., Saab, K., and Ré, C. Structured state spaces: Combining continuous-time, recurrent, and convolutional models, January 2022b. URL https://hazyresearch.stanford.edu/blog/2022-01-14-s4-3. Blog post accessed January 31, 2024.

Hao, S., Angluin, D., and Frank, R. Formal language recognition by hard attention transformers: Perspectives from circuit complexity. *TACL*, 10:800–810, 2022.

Heim, I. File change semantics and the familiarity theory of definiteness. *Semantics Critical Concepts in Linguistics*, pp. 108–135, 1983.

Hesse, W. Division is in uniform TC^0. In *International Colloquium on Automata, Languages, and Programming*, pp. 104–114, 2001.

Hesse, W., Allender, E., and Barrington, D. A. M. Uniform constant-depth threshold circuits for division and iterated multiplication. *J. Comput. Syst. Sci.*, 65:695–716, 2002.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, nov 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

Immerman, N. and Landau, S. The complexity of iterated multiplication. In *[1989] Proceedings. Structure in Complexity Theory Fourth Annual Conference*, pp. 104–111, 1989. doi: 10.1109/SCT.1989.41816.

Kim, N. and Schuster, S. Entity tracking in language models. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3835–3855, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.213. URL https://aclanthology.org/2023.acl-long.213.

Krohn, K. and Rhodes, J. Algebraic theory of machines. i. prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=De4FYqjFueZ.

McKenzie, P., Péladeau, P., and Therien, D. NC1: The automata-theoretic viewpoint. *Computational complexity*, 1:330–359, 1991. ISSN 1016-3328,1420-8954. doi: 10.1007/bf01212963. URL https://link.springer.com/article/10.1007/BF01212963#citeas.

Mereghetti, C. and Palano, B. Threshold circuits for iterated matrix product and powering. *RAIRO-Theor. Inf. Appl.*, 34(1):39–46, 2000. doi: 10.1051/ita:2000105. URL https://doi.org/10.1051/ita:2000105.

Merrill, W. Sequential neural networks as automata. In Eisner, J., Gallé, M., Heinz, J., Quattoni, A., and Rabusseau, G. (eds.), *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pp. 1–13, Florence, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-3901. URL https://aclanthology.org/W19-3901.

Merrill, W. and Sabharwal, A. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11, 2023a. URL https://aclanthology.org/2023.tacl-1.31.

Merrill, W. and Sabharwal, A. A logic for expressing log-precision transformers. In *NeurIPS*, 2023b.

Minsky, M. Neural nets and the brain-model problem. *Unpublished doctoral dissertation, Princeton University, NJ*, 1954.

Mohri, M. *Weighted Automata Algorithms*, pp. 213–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-01492-5. doi: 10.1007/978-3-642-01492-5_6. URL https://doi.org/10.1007/978-3-642-01492-5_6.

Reif, J. H. and Tate, S. R. On threshold circuits and polynomial computation. *SIAM Journal on Computing*, 21(5):896–908, 1992. doi: 10.1137/0221053. URL https://doi.org/10.1137/0221053.

Strobl, L., Merrill, W., Weiss, G., Chiang, D., and Angluin, D. Transformers as recognizers of formal languages: A survey on expressivity, 2023.

Wang, J., Gangavarapu, T., Yan, J. N., and Rush, A. M. Mambabyte: Token-free selective state space model, 2024.

A. Floating-Point Arithmetic

Definition A.1 (Iterated $\mathbb{D}$-product)**.** For $\phi_1, \dots, \phi_z \in \mathbb{D}$, their *iterated* $\mathbb{D}$*-product* is:

$$\bigotimes_{i=1}^{z} \phi_i \triangleq \mathsf{cast}_{\mathbb{D}} \left(\prod_{i=1}^{z} \mathsf{cast}_{\mathbb{R}}(\phi_i) \right). \tag{8}$$

Here $\mathsf{cast}_{\mathbb{R}}$ converts a number in $\mathbb{D}$ to the corresponding real number. $\mathbb{D}$ is implicit in the notations $\mathsf{cast}_{\mathbb{R}}$ and $\bigotimes$.

Lemma A.2 (Iterated $\mathbb{D}$-product)**.** *Let $\phi_1, \dots, \phi_z \in \mathbb{D}$ be such that $z \leq n$ and each ϕ_i can be represented as an n-bit integer. If operators $\mathsf{cast}_{\mathbb{D}}$ and $\mathsf{cast}_{\mathbb{R}}$ are in L-uniform TC^0, then the iterated $\mathbb{D}$-product $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0.*

Proof. By preconditions of the lemma, we can compute $y_i = \mathsf{cast}_{\mathbb{R}}(\phi_i)$ for each i in L-uniform TC^0. Since each ϕ_i is equivalent to an n-bit integer, y_i can be viewed as an n-bit integer. The iterated integer product $y = \prod_{i=1}^{z} y_i$ can be computed with an L-uniform TC^0 circuit (Hesse, 2001). Finally, by a precondition of the lemma, we can cast the result back to $\mathbb{D}$, i.e., compute $\mathsf{cast}_{\mathbb{D}}(y)$ which equals the iterated $\mathbb{D}$-product $\bigotimes_{i=1}^{z} \phi_i$, with an L-uniform TC^0 circuit. □

Lemma A.3 (Iterated float product)**.** *Let $\phi_1, \dots, \phi_z$ be $c \log n$ precision floats and $z \leq n$. Then the iterated float product $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0.*

Proof. The idea is to convert (by scaling up) the sequence of ϕ_i to another sequence of floats that are all representable as integers, apply Lemma A.2, reverse the scaling, and cast the result back to a $c \log n$ precision float.

Let e be the smallest exponent across all ϕ_i and $q = \max\{0, -e\}$. Construct re-scaled floats $\psi_i = \phi_i 2^q$ by adding q to the exponent of ϕ_i, using up to $c \log n$ additional bits in the exponent if necessary to keep the computation exact. Note that e, q, and all ψ_i can easily be computed exactly by an L-uniform TC^0 circuit as they involve fixed-arity arithmetic operations. Further, by construction, every ψ_i has a non-negative exponent and thus represents an integer.

The maximum number representable by each $c \log n$ precision float ϕ_i is upper bounded by 2^{n^c}. Thus, the maximum number representable by each entry ψ_i is $2^{n^c} \times 2^q = 2^{n^c+q}$. Let $m = n^c + q$. It follows that each ψ_i can be equivalently represented as an m-bit integer. Further, this integer can be computed by left-shifting the mantissa of ψ_i by a number of bits equal to the value of the exponent of ψ_i (which is non-negative). Finally, this left-shift, and thus the $\mathsf{cast}_\mathbb{R}$ operation over m-precision floats, can be easily computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. In the other direction, casting from reals to m-precision floats can also be easily accomplished by an L-uniform threshold circuit of size $\mathrm{poly}(m)$.

Observing that $\psi_1, \ldots, \psi_z$ is a sequence of floats each representable as an m-bit integer, we now apply Lemma A.2 with $\mathbb{D}$ being 'float' to conclude that iterated float product $\tau = \bigotimes_{i=1}^{z} \psi_i$ can be computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. Since $m \leq 2n^c$, this circuit is also of size $\mathrm{poly}(n)$.

Finally, to compute the original iterated float product $\bigotimes_{i=1}^{z} \phi_i$, we divide τ by 2^{qz}. This can be accomplished by subtracting qz from the exponent of τ; again, we do this computation exactly, using up to $(c+1) \log n$ additional bits in the exponent if necessary. We then cast the resulting float back to a $c \log n$ precision float. All this can be done in L-uniform TC^0, finishing the proof that $\bigotimes_{i=1}^{z} \phi_i$ can be computed in L-uniform TC^0. □

We now extend these notions and results to matrix powering.

Definition A.4 ($\mathbb{D}$-matrix power)**.** For a matrix M over $\mathbb{D}$ and $z \in \mathbb{Z}^+$, $\mathbb{D}$-*matrix power* is defined as

$$M^z \triangleq \mathsf{cast}_\mathbb{D}(\mathsf{cast}_\mathbb{R}(M)^z). \tag{9}$$

Mereghetti & Palano (2000) showed that when the datatype $\mathbb{D}$ is n-bit integers, one can compute M^n in TC^0. We note that their construction also works for computing M^z for any $z \leq n, z \in \mathbb{Z}^+$. Further, as they remark, their construction can, in fact, be done in *uniform* TC^0. Specifically, we observe most of their construction involves sums and products of constantly many n-bit integers, which can be done in L-uniform TC^0. The only involved step is dividing a polynomial of degree (up to) n by a polynomial of degree (up to) $k-1$ and returning the remainder. It turns out that this 'polynomial division with remainder' operation can also be performed in L-uniform TC^0 (see Corollary 6.5 of Hesse et al., 2002 and an explanation in Appendix A.1). We thus have the following extension of Mereghetti & Palano's result:

Lemma A.5 (Integer matrix power, derived from Mereghetti & Palano, 2000)**.** *Let $k \in \mathbb{Z}^+$ be a fixed constant. Let M be a $k \times k$ matrix over n-bit integers and $z \leq n, z \in \mathbb{Z}^+$. Then integer matrix power M^z can be computed in L-uniform TC^0.*

We extend this result to matrix powers over $\mathbb{D}$ rather than integers:

Lemma A.6 ($\mathbb{D}$-matrix power)**.** *Let $k \in \mathbb{Z}^+$ be a fixed constant. Let M be a $k \times k$ matrix over a datatype $\mathbb{D}$ with entries equivalently representable as n-bit integers. Let $z \leq n, z \in \mathbb{Z}^+$. If operators $\mathsf{cast}_\mathbb{D}$ and $\mathsf{cast}_\mathbb{R}$ are in L-uniform TC^0, then $\mathbb{D}$-matrix power M^z can be computed in L-uniform TC^0.*

Proof. By preconditions of the lemma, we can compute $\mathsf{cast}_\mathbb{R}(M)$ in L-uniform TC^0. Since the entries of M are equivalent to n-bit integers, $\mathsf{cast}_R(M)$ can be viewed as a $k \times k$ integer matrix of n-bit integers. By Lemma A.5, we can compute $\mathsf{cast}_R(M)^z$ using an L-uniform TC^0 circuit. Finally, by a precondition of the lemma, we can cast the result back to $\mathbb{D}$, i.e., compute $\mathsf{cast}_\mathbb{D}(\mathsf{cast}_\mathbb{R}(M)^z)$ which equals M^z, with an L-uniform TC^0 circuit. □

Lemma A.7 (Float matrix power)**.** *Let $k, c \in \mathbb{Z}^+$ be fixed constants. Let M be a $k \times k$ matrix over $c \log n$ precision floats. Let $z \leq n, z \in \mathbb{Z}^+$. Then float matrix power M^z can be computed in L-uniform TC^0.*

Proof. The idea is to convert (by scaling up) M to another float matrix all whose entries are representable as integers, apply Lemma A.6, reverse the scaling, and cast the result back to $c \log n$ precision floats.

Let e be the smallest exponent across all float entries of M and $q = \max\{0, -e\}$. Construct a re-scaled float matrix $\tilde{M} = M 2^q$ by adding q to the exponent of every entry of M, using up to $c \log n$ additional bits in the exponent if necessary to keep the computation exact. Note that e, q, and $\tilde{M}$ can easily be computed exactly by an L-uniform TC^0 circuit as they involve fixed-arity arithmetic operations. Further, by construction, $\tilde{M}$ has non-negative exponents in all its float entries. Thus, every entry of $\tilde{M}$ represents an integer.

The maximum number representable by each $c \log n$ precision float in M is upper bounded by 2^{n^c}. Thus, the maximum number representable by each entry of $\tilde{M}$ is $2^{n^c} \times 2^q = 2^{n^c+q}$. Let $m = n^c + q$. It follows that each entry ϕ of $\tilde{M}$ can be equivalently represented as an m-bit integer. Further, this integer can be computed by left-shifting the mantissa of ϕ by a number of bits equal to the value of the exponent of ϕ (which is non-negative). Finally, this left-shift, and thus the $\mathsf{cast}_{\mathbb{R}}$ operation over m-precision floats, can be easily computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. In the other direction, casting from reals to m-precision floats can also be easily accomplished by an L-uniform threshold circuit of size $\mathrm{poly}(m)$.

Note that $2^q \in [0, n^c]$ and hence $m \in [n^c, 2n^c]$. In particular, $m \geq n$. Thus $z \leq n$ (a precision) implies $z \leq m$. Observing that $\tilde{M}$ is a matrix of floats each representable as an m-bit integer, we now apply Lemma A.6 with $\mathbb{D}$ being 'float' to conclude that float matrix power $\tilde{M}^z$ can be computed by an L-uniform threshold circuit of size $\mathrm{poly}(m)$. Since $m \leq 2n^c$, this circuit is also of size $\mathrm{poly}(n)$.

Finally, to compute M^z, we first divide each entry of $\tilde{M}^z$ by 2^{qz}. This can be accomplished by subtracting qz from the exponent of each entry of $\tilde{M}$; again, we do this computation exactly, using up to $(c+1)\log n$ additional bits in the exponent if necessary. We then cast all entries of the resulting matrix back to $c \log n$ precision floats. All this can be done in L-uniform TC^0, finishing the proof that M^z can be computed in L-uniform TC^0. □

A.1. L-Uniformity of Polynomial Division in TC^0

Hesse et al. (2002) state that polynomial division is in L-uniform TC^0 in Corollary 6.5. For historical reasons, this claim is preceded by weaker claims in older papers. We briefly clarify this situation to show why the stronger claim is valid.

Reif & Tate (1992) establish that polynomial division can be done in P-uniform TC^0, whereas we state our results for L-uniform TC^0. However, the only issue preventing these results from going through in the L-uniform case is that, at the time of publication, it was not known whether integer division and iterated integer multiplication were computable in L-uniform TC^0. However, Hesse (2001) later proved exactly this. Combining the two results, Theorem 3.2 from Reif & Tate (1992) goes through with L-uniformity. Its Corollary 3.3 then allows us to conclude that integer polynomial division can be solved by L-uniform TC^0 circuits because the output of integer polynomial division is an analytic function whose Taylor expansion has a finite number of terms (Reif & Tate, 1992).

B. Chess Reduction

We let $\mathcal{M}$ denote the set of chess moves in (source square, target square) notation.

Proposition B.1. *S_5 can be reduced to chess state tracking via NC^0 reductions.*

Proof. Without loss of generality, we consider the variant of S_5 where the output is true if and only if the original first element returns to the first position after the given sequence of permutations has been applied. Given an S_5 instance, we will construct an initial board state and sequence of moves such that the final chessboard state encodes the output of the S_5 problem instance.

Initial Board State. We construct a chessboard similar to Figure 1 but with a black rook at a8 and black queens at b8 to e8.

Chess Move Sequence. We then construct a finite function $f : S_5 \to \mathcal{M}^*$ that encodes a permutation π as a sequence of chess moves. We first factor each permutation π to a sequence of transpositions $\tau_1(\pi) \cdots \tau_{m_\pi}(\pi)$. Each transposition τ can in turn be expressed as a sequence of chess moves analogously to Figure 1. For example, transposing items 1 and 3 can be expressed as the move sequence: (a8, a7), (a1, b1), (c8, c6), (b1, a1), (a7, c7), (a1, b1), (c6, a6), (b1, a1), (c7, c8), (a1, b1), (a6, a8), (b1, a1), which has the crucial property that it transposes a8 with c8. We denote the mapping from transpositions to chess move sequences as $f : \mathcal{T} \to \mathcal{M}^*$. Putting it all together, we have that

$$f(\pi) = \prod_{j=1}^{m_\pi} f(\tau_j(\pi)).$$

To reduce a sequence of permutations $w \in S_5^*$, we let

$$f(w) = \prod_{i=1}^{n} f(w_i).$$

Putting It All Together. We call our oracle for chess state tracking with the constructed initial board state and $f(w)$ as the sequence of chess moves. By construction, we can then return true if and only if the rook is at a8. The reduction can be implemented in NC^0 because it is a simple elementwise mapping of the input tokens, and decoding from the output chessboard is a finite table lookup. □

As a fun aside, we note that the chess board constructed in Proposition 3.3 is reachable in a standard chess game. The chess sequences encoding permutation sequences are all valid chess games, except that they ignore the fact that repeated board states in chess will technically lead to draws.