Electronic Colloquium on Computational Complexity, Report No. 14 (2013)

Fast Algorithms for Interactive Coding

Zvika Brakerski[*] Moni Naor[†]

Abstract

Consider two parties who wish to communicate in order to execute some interactive protocol π. However, the communication channel between them is noisy: An adversary sees everything that is transmitted over the channel and can change a constant fraction of the bits as he pleases, thus interrupting the execution of π (which was designed for an errorless channel). If π only contains a single long message, then a good error correcting code would overcome the noise with only a constant overhead in communication. However, this solution is not applicable to *interactive protocols* consisting of many short messages.

Schulman (FOCS 92, STOC 93) presented the notion of *interactive coding*: A simulator that, given any protocol π, is able to simulate it (i.e. produce its intended transcript) even with constant rate adversarial channel errors, and with only constant (multiplicative) communication overhead. Until recently, however, the running time of all known simulators was exponential (or sub-exponential) in the communication complexity of π (denoted N in this work). Brakerski and Kalai (FOCS 12) recently presented a simulator that runs in time $\text{poly}(N)$. Their simulator is randomized (each party flips private coins) and has failure probability roughly 2^{-N}.

In this work, we improve the computational complexity of interactive coding. While at least N computational steps are required (even just to output the transcript of π), the BK simulator runs in time $\tilde{\Omega}(N^2)$.

We present two efficient algorithms for interactive coding: The first with computational complexity $O(N \log N)$ and exponentially small (in N) failure probability; and the second with computational complexity $O(N)$, but failure probability $1/\text{poly}(N)$. (Computational complexity is measured in the RAM model.)

[*]Stanford University, `zvika@stanford.edu`. Supported by a Simons Postdoctoral Fellowship and by DARPA.

[†]Dept. of Computer Science and Applied Math, Weizmann Institute of Science, `moni.naor@weizmann.ac.il`. Research supported in part by a grant the Israel Science Foundation.

ISSN 1433-8092

1 Introduction

Communication over a noisy channel is a fundamental issue in computer science, engineering and related fields. Shannon [Sha48] and Hamming [Ham50] initiated the modern study of error correcting codes, which continues to be a thriving area to these days. A good (asymptotic) error correcting code encodes a k-bit message into an $O(k)$-bit codeword, such that an adversarial change to at most a δ-fraction of the bits of the codeword (for a constant δ), still enables decoding the original message. A landmark in this area is the work of Justesen [Jus72] who showed the first explicit construction of such good codes.

Among other parameters (such as the information overhead and the allowed error rate), the *computational complexity* of error correcting codes has been the focus of extensive research. The usability of the code, both practically and theoretically, depends on the ability to encode and decode efficiently. This research effort culminated in the work of Spielman [Spi95] who showed that good codes can be encoded and decoded in *linear* time in the RAM model. Followup works (like [GI05]) improved the parameters of this result, getting near optimal rates.

Error correcting codes, however, fall short in shielding *interactive protocols* against channel errors: Consider two parties who wish to execute a multiple-message protocol, say each party sends one bit at a time, and the channel is noisy. Using error correcting code on each message at a time will obviously not protect against an adversary that can change a constant fraction of the communication over the channel. This motivated Schulman [Sch92, Sch93] to present the notion of *interactive coding*. An interactive coding scheme is a simulator algorithm S such that given any interactive protocol $\pi = (A, B)$ (where A, B are interactive machines), (S^A, S^B) is a protocol which outputs the transcript of π (and thus allows to compute whatever it was that π computed), even when executed over a channel with constant adversarial error rate. Furthermore, we require that the communication complexity of S^π is linearly related to that of the original π.

Schulman showed that interactive coding was achievable for an error rate of roughly 1/240, but his construction relied on a combinatorial object called a *tree code*. Schulman showed how to construct and decode tree codes in exponential time, yielding a simulator S^π whose computational complexity is $2^{O(N)}$, where N is the length of the transcript of π. Followup works by Braverman and Rao [BR11] and by Braverman [Bra12] showed how to improve the error rate to 1/8 (1/4 for non-binary alphabet) and how to improve the computational complexity to $2^{O(N^\epsilon)}$, respectively. Gelles, Moitra and Sahai [GMS11] showed that the computational complexity can be improved to poly(N) if the channel errors are uniformly distributed (i.e. each bit is flipped with the same fixed probability), however their simulator still required exponential time for adversarial channels. Recently, Brakerski and Kalai [BK12] showed how achieve error rate 1/32 with computational complexity poly(N). Their idea was to divide the transcripts into chunks of logarithmic length, apply the previous exponential-time algorithms to the chunks, and use randomness-efficient hashing to check for consistency and eliminate chunks that were corrupted by the adversary (see a more detailed outline of this work below). The use of hashing meant that some failure probability was introduced, specifically [BK12] showed that for any adversary, the failure probability is at most $2^{-\Omega(N)}$.

In this work, we further improve the computational complexity of interactive coding, and show that exponentially low failure probability is achievable with almost-linear time simulators (running in time $O(N \log N)$); and that if we only require polynomially low failure probability, then a linear time simulator exists. This is shown for an unspecified constant error rate, however if polynomial (protocol independent) preprocessing is allowed, the error rate of [BK12] can be matched. See below for details.

1.1 Our Results

We present two efficient algorithms for simulating interactive protocol in the presence of constant rate adversarial noise. Our simulators are implemented as a RAM machine with logarithmic words and oracle access to the original protocol (the machines A, B).

Our first algorithm, presented in Section 3, runs in time $O(N \log N)$ (recall that N is the communication complexity of the original protocol) and succeeds with all but exponentially small probability:

Theorem A. *There exists an interactive simulator against constant rate adversarial error with communication complexity $O(N)$, failure probability $2^{-\Omega(N)}$ and computational complexity $O(N \log N)$.*

An interesting feature of this algorithm is that the failure probability can be brought down to $2^{-\tau N}$ for *any* constant τ (at the cost of a constant-factor increase in the communication complexity). This property is also implicit in the [BK12] algorithm, though not pointed out there.

Our second algorithm, presented in Section 4, runs in time $O(N)$ (which is optimal, up to a constant), but fails with probability $1/\text{poly}(N)$, for an arbitrarily chosen polynomial:

Theorem B. *There exists an interactive simulator against constant rate adversarial error with communication complexity $O(N)$, failure probability $1/\text{poly}(N)$ and computational complexity $O(N)$.*

Both of our algorithms are proven to be robust against some (unspecified) constant error rate, this is in contrast to previous works who provided explicit rates (e.g. 1/32 in [BK12]). We remark that we can match the rate of [BK12] if polynomial time preprocessing is allowed.

1.2 Our Techniques

We will briefly sketch the simulator from [BK12] (henceforth BK), which is the starting point of our work. BK use the exponential time simulator from previous works as a building block. In order to run in polynomial time in N, they only execute this simulator on inputs of logarithmic length. They thus divide the transcript of the original protocol π into chunks of length $\log N$ and attempt to simulate them one after the other. This approach, as is, cannot work, since the adversary might concentrate many errors towards interrupting the simulation of an early chunk, thus making the parties lose synchronization. BK therefore add a "header" to each chunk, in which each of the communicating parties sends a hash of their local transcript as recovered so far (denoted T), as well as the length of T. If the parties see that they are out of sync, i.e. that their local transcripts differ in length or contents, they "roll back" and remove a chunk from their local transcript. This process is shown to converge into the parties regaining agreement. The header length is $O(\log N)$ and therefore the total length of header plus chunk is also $O(\log N)$. Communicating this information using the inefficient simulator can be done, therefore, in $\text{poly}(N)$ steps. While the use of hashing introduces an error, the choice of the hash function for each round of communication is independent, and therefore concentration bounds imply that the number of hash faults will be low with all but exponentially small probability.

The computational complexity of the BK simulator comes from two main ingredients: First, the exponential time simulator introduces some unspecified $\text{poly}(N)$ computational complexity in each round of the protocol. Second, and perhaps more importantly, the BK algorithm requires the parties to hash (all) their local state T before communicating each chunk. Since the length of T will quickly grow into $\Omega(N)$ bits, the total computational complexity of hashing throughout the protocol is $\tilde{\Omega}(N^2)$.

Linear Time Simulation with Exponential Preprocessing. To solve the first problem, we observe that the exponential time simulators of [Sch96, BR11] can be made to run in *linear* time, given exponential-time preprocessing. Furthermore, this preprocessing does not depend on the specific protocol being simulated, only on its communication complexity. This means that if we chose our chunk size to be small enough, i.e. some $\gamma \log N$ for a small constant γ, the preprocessing will run in time $O(N)$ and each chunk will be simulated in time $O(\log N)$, which will bring the total computational overhead of chunk simulation to the desired $O(N)$. In a nutshell, the observation is that the previous simulators invest exponential time in decoding the transcript of the simulation so far, and this decoding is independent of the specific protocol. Therefore, given that our simulation is of logarithmic length, we can create a table (or better, a decision tree) with the decodings of all possibilities, which will enable linear-time simulation.

This solution, however, has an unfortunate implication: The header that contains the synchronization information cannot be made as short as we wish, and its size is too large for to allow linear time preprocessing. This is solved by noticing that the header is in fact non-interactive and therefore can be encoded by a standard error correcting code with linear-time encoding and decoding. The outcome is that our simulator inherits the error rate from the error correcting code - hence the unspecified constant error rate.

As we mentioned above, allowing polynomial time preprocessing will allow to incorporate the header into the simulator as is done in BK, and match their error rate.

Efficient Segmented Hashing and the $O(N \log N)$ Solution. In order to make the hashing process more efficient, we notice that randomness efficient hashing can be viewed as nothing more than encoding the input using an error correcting code, and then outputting specific locations in this codeword according to a random walk on an expander (hence the randomness efficiency compared to sampling random locations). The computational complexity comes mostly from the encoding of the input, which is independent of the randomness of the hash function.

In our case, the local transcript T, which is the input to the hash, changes very slowly during the course of the algorithm. This means that consecutive applications of the hash function will have almost the same input, up to a single $O(\log N)$-length chunk. We take advantage of this property by dividing T into segments, and encoding each segment using a linear-time error correcting code. We start encoding a segment only after it was received in full, and encode lazily, at a pace that is proportional to the communication over the channel.[1] Then, when we need to hash the entire transcript T, we will find a subset of the segments that have finished being encoded and that cover the entire transcript, and evaluate the hash to each of their respective codewords (that is, probe the codewords in the appropriate locations according to the hash function description).

But how long should the segments be? On one hand, if the segments are too short, then many of them will be needed in order to cover the entire transcript, which would make the evaluation step too expensive (e.g. constant size segments will require evaluating the hash function a linear number of times). On the other hand, if they are too long, then the encoding might not be ready when we need it (since our encoding is lazy, the codeword will only be ready after the end of the following segment). Our solution, therefore, is to encode in parallel segments of all sizes (in logarithmic scale). This means that our computational overhead is logarithmic, since each bit of the transcript is encoded in a logarithmic number of codewords, and this will also guarantee that the transcript can be covered by a logarithmic number of segments.

At each round of the protocol, we will generate two hash functions: one will be evaluated on the encodings of all segments (using the union bound we will show that the probability of failure remains small). We will have $O(\log N)$ segments, each producing $O(\log N)$ bits of hash value. The total of

[1]Lazy evaluation is required since sometimes the protocol will roll back and erase a part of the transcript, and we don't want too much work to go to waste.

$O(\log^2 N)$ bits will still be too long to send on the channel (note that the hash value is an overhead on top of an $O(\log N)$ long chunk). Therefore, we will encode it on the fly using our efficient error correcting code, and evaluate the second hash function on it. This will produce an $O(\log N)$ hash value with the desired properties.

Working with Hashed History and the $O(N)$ Solution. To achieve linear running time, we use a different idea for hashing. When we come to hash a value T, we already have the hash of T's "predecessor" T': a value that is the same as T but without the last chunk. Our linear algorithm will use this value is representative of T' and "forget" about the rest of the history altogether.

Let HH (for "hashed history") denote the hash value that was computed in the previous round, say that this value corresponds to some local transcript T. When we append a new chunk L to T, we will compute a new hash value HH' by applying a new hash function to (essentially) $HH \| L$ (rather than $T \| L$ as in the previous solution). Naturally, computing a hash on such a short value leads to great efficiency improvement and ultimately to a linear-time algorithm.

This approach may seem a little risky, since the hash function has collisions. If the parties' states are not equal but still fall into the same equivalence class (namely hashed into the same value) at even one point in the protocol, then this may never be corrected and the simulation will fail! However, we can choose the hash function so that the probability of collision is at most $1/\text{poly}(N)$ per application of a hash function. Since there are only $O(N)$ rounds in the protocol, the union bound asserts that the error probability is bounded by $1/\text{poly}(N)$.

We are still ignoring a very important factor in the simulation - the adversary who sees everything that is going on over the channel and might potentially create collisions even where they are not supposed to exist. This is handled as follows: The parties will only choose the new hash function after the previous HH was sent over the channel, so in order to create a collision, the adversary must change the hash function (rather than its input). However, if the adversary created errors in the hash function, then the states of the two parties will differ since one of them received the wrong description of the hash function (while the party who generated the hash function of course has the correct value). We thus append the hash function itself as a part of the state that will be compared in the next rounds. This way, the adversary will be unable to make the parties "falsely agree" by inserting errors on the channel, and we go back to the situation that we know how to analyze according to the [BK12] guidelines.

1.3 Cryptography and Interactive Coding

The problem of interactive coding is non-cryptographic in nature, however concepts and techniques from cryptography have clearly influenced both this work as well as the BK work. One important idea is using a succinct authentication protocol in order to check the previous transcript. Then other inspirations come from the work on incremental cryptography [BGG94] and memory checking [BEG$^+$94] where the goal is to perform some sort of authentication without rereading the full file/memory. The tricky part is where to embed the secret key, since in this setting there are no shared secrets or a shared trusted key with a corresponding secret key.

2 Preliminaries

2.1 Interactive Protocols

An interactive protocol $\pi = (A, B)$ is a pair of interactive machines. Each machine implements a function $\{0,1\}^* \to \{0,1\}$ such that on an input $T \in \{0,1\}^*$ (think about this as the transcript of the communication so far) the machine outputs $A(T)$ (alternatively $B(T)$) which is the next bit to be transmitted. We define $T_0 = \phi$, and for every even i we let $T_i = T_{i-2} \| A(T_{i-2}) \| B(T_{i-2})$. The *transcript* of π, denoted $\mathsf{Trans}(\pi)$ is the string T_N for an even N which we call the *communication complexity* of π, and also denote by $\mathsf{CC}(\pi)$. We choose to view A, B as machines that can compute indefinitely rather than ones that decide to break after N steps.

We note that one can consider other communication models: for example to allow multi-bit messages at every round, or to define the rounds adaptively: $T_i = T_{i-2} \| A(T_{i-2}) \| B(T_{i-2} \| A(T_{i-2}))$. We chose to present the simplest possible model, but our results extend to the aforementioned models as well.

In this work, we consider simulators for interactive communication. A simulator produces a new protocol, that uses the original protocol as an oracle, and computes the transcript of the original protocol. It is sufficient to simulate deterministic protocols with no input, since we can always hardwire the randomness and input into the protocol.

2.2 Computational Model

In this paper we consider computational complexity of interactive protocol simulators. Formally, our computational model is a RAM machine with logarithmic (in the communication complexity) word length, and with oracle access to an interactive machine X. The communication with the oracle is using a designated area in the memory. Writing T in this designated area and making an oracle call, will write next message $X(T)$ into a (different) area in the memory. Note that the complexity of appending (or truncating) information to (or from) the end of T can be done in linear time in the length of the additional part (or subtracted part) regardless of the length of T.

Preprocessing. We say that an algorithm A has running time t with preprocessing t' if there exists a preprocessing algorithm $\mathsf{PreProc}_A$ that runs in time t' and produces some output z, such that A^z (A with RAM access to z) runs in time t. (Note that $|z| \leq t'$.) We will usually omit the superscript z where it is clear from the context.

2.3 Linear Time ECC and Randomness-Efficient Hashing

Essential building blocks for our constructions are linear-time encodable and decodable error correcting codes, as well as randomness and computation efficient hash functions. A required ingredient for those are efficiently computable expanders.

Theorem 2.1 (implicit in [RVW00])**.** *For every ϵ there exist a constant $d \in \mathbb{N}$, and an algorithm $\mathsf{ExpGen} = \mathsf{ExpGen}_\epsilon$ such that $\mathsf{ExpGen}(N)$ outputs an N-vertex d-regular graph whose second eigenvalue is smaller than ϵd. Furthermore, ExpGen runs in time $O(N)$.*

Using the linear-time constructible expanders described above, one can achieve linear time encodable and decodable error correcting codes.

Theorem 2.2 ([Spi95, GI05])**.** *There exists a family $\mathcal{C} = \{\mathcal{C}_k\}_{k \in \mathbb{N}}$ of error correcting codes with information rate $r > 0$ and error rate $\delta > 0$, where the encoding and decoding procedures $\mathsf{FastEnc} : \{0,1\}^k \to \{0,1\}^{k/r}$, $\mathsf{FastDec} : \{0,1\}^{k/r} \to \{0,1\}^k$ run in linear time.*

Randomness-Efficient Hash Functions with Preprocessing. To boost the efficiency of the hashing process, we notice that it can be divided into three different tasks with fairly weak dependence:[2]

[2]This was previously used, e.g. in [NN90].

The hashing process will start by encoding the input with a good error correcting code (this part depends only on the input); then it will decide on a set of indices in the codeword (this process will only depend on the description of the hash function and not on the codeword itself); lastly it will access the codeword in exactly those indices. The output of the hashing process is the set of values assigned by the code to the set of selected indices.

Intuitively, this process will achieve good hashing properties since two different inputs will produce two codewords that differ in a constant fraction of locations. In that case, a random set of indices will have a good chance of hitting an index where the codewords differ (the probability of error decreases exponentially with the number of samples). To save on randomness, we use a good disperser in the form of a random walk over an expander, instead of using completely random samples.

For our error correcting code, we will use the code $\mathcal{C}$ from Theorem 2.2. We note that the encoding process is completely independent of the hash function to be applied. The expander random walk will be implemented using the expander from Theorem 2.1. This part, in turn, is completely independent of the input to the hash function.

The following theorem summarizes the properties of the mapping between the description of a hash function and a set of indices. It follows from standard randomness efficiency arguments (see, e.g., survey in [Gol97]).

Theorem 2.3. *There exists a constant $q > 1$ and a function $\mathsf{HashMap}_{N,\epsilon}(k, \mathsf{rand})$ that given RAM access to an N-node d-regular expander with constant spectral gap (as in Theorem 2.1) and given inputs $k|N$, $\epsilon \in (0,1)$ and random string $\mathsf{rand} \in \{0,1\}^{q\log(N/\epsilon)}$, produces a set of indices $I \subseteq [k]$ of cardinality $O(\log(1/\epsilon))$ with the following property: Let $x, y \in \{0,1\}^k$ with relative hamming distance $\geq \delta$ (where δ is as in Theorem 2.2), then $\Pr_{\mathsf{rand}}\big[x[I] = y[I]\big] \leq \epsilon$ (where $x[I]$ denotes the vector whose coordinates are $x[i]$ for all $i \in I$). Furthermore, $\mathsf{HashMap}$ runs in time $O(\log(N/\epsilon))$.*

The following corollary is therefore immediate.

Corollary 2.4. *There exists a constant $q > 1$ (the same as in Theorem 2.3) and a function $\mathsf{LinHash}_\epsilon(x, \mathsf{rand})$ that given $x \in \{0,1\}^k$ and random string $\mathsf{rand} \in \{0,1\}^{q\log(k/\epsilon)}$, runs in linear time and produces an $O(\log(1/\epsilon))$ bit output, such that for all $x \neq y$*

$$\Pr_{\mathsf{rand}}\big[\mathsf{LinHash}_\epsilon(x, \mathsf{rand}) = \mathsf{LinHash}_\epsilon(y, \mathsf{rand})\big] \leq \epsilon\ .$$

2.4 Inefficient Interactive Simulators

An essential building block in our construction are the inefficient (exponential-time) simulators from previous works. Since we aim for a linear time simulation (or almost linear time), we cannot afford any of our components to run in such prohibitive time. What we do take from these simulators is that with an appropriate preprocessing and space, we can actually execute the simulator in linear time. This will be sufficient for our purposes. The following theorem is implicit in previous works.

Theorem 2.5 (implicit in [Sch96, BR11, Bra12])**.** *There exist positive constants $\rho, \eta \in (0,1)$ and a deterministic interactive oracle machine ExpSim (the simulator) such that for any protocol $\pi = (A, B)$ of communication complexity $n = \mathsf{CC}(\pi)$, the protocol $\mathsf{ExpSim}^\pi = (\mathsf{ExpSim}^A, \mathsf{ExpSim}^B)$ computes $\mathsf{Trans}(\pi)$, has communication complexity $\mathsf{CC}(\mathsf{ExpSim}^\pi) \leq \mathsf{CC}(\pi)/\rho$, and is robust (with probability 1) to adversarial error of rate η. The computational complexity of $\mathsf{ExpSim}^{(\cdot)}$ is $O(n)$ with preprocessing $O(2^{n/\gamma})$ for some constant $\gamma > 0$ (where the preprocessing depends only on n and not on A, B).*

Proof. In both [Sch96, BR11], the simulator works in the following way: It first examines everything that was broadcast over the channel until now (the information that it sent and received). This information is decoded (based on a tree code) into a view as to the internal state of both parties. From the decoded view, the simulator can produce two things: A query to the oracle A/B and a predicate that determines what is the next symbol to be sent based on the oracle's answer.

The critical observation is that this entire process only depends on the information that was sent over the channel since the beginning of the simulation, and not on the protocol (A, B). The only dependence on the protocol is when making the oracle call.

This process can be modeled by a decision tree whose input is the information communicated so far and its outputs are the appropriate oracle query and the predicate. This decision tree does not depend on the protocol and can be manufactured offline once and for all. Given RAM access to this decision tree, the simulation process of any n-bit protocol only requires linear time. The preparation of the decision tree can take exponential time in n. $\square$

Since the preprocessing time as well as the space to represent the result are exponential we really cannot apply this theorem on large chunks, but rather on blocks which are roughly logarithmic in the total communication.

3 $O(N \log N)$-Time Simulator with Exponentially Small Error

In this section we present a simulator for interactive protocols that can withstand a constant fraction of errors and whose computational overhead is only slightly higher than that of the original protocol. Specifically, the oracle-RAM complexity of our protocol is $O(N \log N)$, compared to the trivial lower bound of $O(N)$. Our simulator is randomized and succeeds with probability $2^{-\tau N}$ where τ can be *any* constant. Namely, we can reduce the error term as much as we wish. For overview of this solution, see Section 1.2.

In this section we present a simulator for interactive protocols that can withstand a constant fraction of errors and whose computational overhead is only slightly higher than that of the original protocol. Specifically, the oracle-RAM complexity of our protocol is $O(N \log N)$, compared to the trivial lower bound of $O(N)$. Our simulator is randomized and succeeds with probability $2^{-\tau N}$ where τ can be *any* constant. Namely, we can reduce the error term as much as we wish. For overview of this solution, see Section 1.2.

Consider a protocol π with communication complexity $\mathsf{CC}(\pi) = N$. Define the chunk length of our protocol to be $\ell_{\mathsf{chunk}} \triangleq \lfloor \gamma \log N \rfloor$, where γ is as defined in Theorem 2.5.

Our simulator $\mathsf{QLinSim}$ is described in Figures 1, 2 and 3. Figure 1 is the main procedure; Figure 2 describe the subroutine for hashing and Figure 3 describes "threads": subroutines with an internal state. Unlike a subroutine that is called with a certain input, executes, and returns an output, a thread never terminates: it reads its input and writes its output to shared memory. When the main procedure calls a thread, it specifies the number of computational steps that the thread should perform, the thread runs for that number of steps and returns the control to the main procedure.

The simulator is parameterized by a constant c whose value will be determined in the analysis, where we assume w.l.o.g that cN/ℓ_{chunk} is a power of 2; and by a parameter $\epsilon = 1/\mathrm{poly}(N)$ that enables to control the failure probability of the simulator (for $\epsilon = N^{-a}$, the failure probability will be bounded by $2^{-\Omega(1)\cdot aN}$).

The simulator $\mathsf{QLinSim}$ makes use of the inefficient simulator ExpSim (Theorem 2.5), the error correcting code $\mathcal{C}$ (Theorem 2.2) and the hash functions $\mathsf{HashMap}, \mathsf{LinHash}$ (Theorem 2.3 and Corollary 2.4).

Simulator $\mathsf{QLinSim}^X$

- **Input:** Oracle access to interactive machine X.
- **Output:** Transcript $T \in \{0,1\}^N$.
- **Operation:**

  0. Preprocessing for the components of the algorithm.
     (a) Run $\mathsf{ExpGen}(cN/r)$ to generate an expander graph (see Theorem 2.1).
     (b) Run the preprocessing procedure $\mathsf{PreProc}_{\mathsf{ExpSim}}$ for the simulator ExpSim, with $n = \ell_{\mathsf{chunk}} = \gamma \log N$ (see Theorem 2.5).
  1. Set $T := \phi$, $i := 0$.
  2. Repeat cN/ℓ_{chunk} times:
     (a) For all $t = 1, \ldots, \log(cN/\ell_{\mathsf{chunk}})$, perform $O(\ell_{\mathsf{chunk}})$ computation steps of Thread_t (see Figure 3).
     (b) Sample $\mathsf{rand}_{1,x}, \mathsf{rand}_{2,x} \leftarrow \{0,1\}^{q \log(cN/\epsilon)}$.
     (c) Define $\sigma_x := \mathsf{TwoLevelHash}(\mathsf{rand}_{1,x}, \mathsf{rand}_{2,x})$, where $\mathsf{TwoLevelHash}$ is defined in Figure 2.
     (d) Encode $(i, \mathsf{rand}_{1,x}, \mathsf{rand}_{2,x}, \sigma_x)$ using $\mathcal{C}$ to obtain $\omega_x := \mathsf{FastEnc}(i, \mathsf{rand}_{1,x}, \mathsf{rand}_{2,x}, \sigma_x)$. See Theorem 2.2 (note that $|\omega_x| = O(\log N)$).
     (e) Send ω_x over the channel and receive a word ω_y of the same length.[a] Decode ω_y to obtain $(j, \mathsf{rand}_{1,y}, \mathsf{rand}_{2,y}, \sigma_y)$.
     (f) Let $\tilde{\sigma}_y := \mathsf{TwoLevelHash}(\mathsf{rand}_{1,y}, \mathsf{rand}_{2,y})$.
     (g) Use the simulator ExpSim to simulate X for ℓ_{chunk} more rounds. The output of ExpSim is a simulated transcript $L \in \{0,1\}^{\ell_{\mathsf{chunk}}}$.
     (h) Proceed according to the following cases:
       - If $(i > j)$ then remove the last chunk from T and set $i := i - 1$.
       - If $(i < j)$ then finish this iteration.
       - If $((i = j)$ and $(\tilde{\sigma}_y \neq \sigma_y))$ then remove the last chunk from T and set $i := i - 1$.
       - If $((i = j)$ and $(\tilde{\sigma}_y = \sigma_y))$ then append the new chunk L onto T and set $i := i+1$.
  3. Output the first N bits of T.

[a]To be more explicit: The party whose turn it is to speak sends their ω first, and then the second party sends their value. If we consider a channel where messages are concurrent then both parties can send at the same time.

Figure 1: An $N \log N$ time simulator.

In the course of the simulation, each party maintains a variable T of length at most cN, corresponding to its local view of the reconstructed transcript. We will consider $O(\log N)$ divisions of T into segments, where the t^{th} division will be into segments of length $2^t \cdot \ell_{\mathsf{chunk}}$ (namely, of the form $T[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} + 1 : (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$). The t^{th} thread will be in charge of (lazily) encoding the segments of the t^{th} division using the code $\mathcal{C}$.

Generally speaking, $\mathsf{QLinSim}$ takes after the [BK12] simulator, with a few important differences:

Subroutine $\mathsf{TwoLevelHash}(\mathsf{rand}_1, \mathsf{rand}_2)$

1. Use a greedy algorithm to "cover" T with segments:
   (a) We say that a segment of the form $[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} + 1 : (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$ is *ready* if $i \geq (k+2) \cdot 2^t$ (recall that $i = |T| / \ell_{\mathsf{chunk}}$).
   (b) Set $z := 0$. Then repeatedly choose the longest segment that starts at $z+1$ and is ready, and set z to be the endpoint of the chosen segment.

   Let s denote the number of segments and let $w_1, \ldots, w_s$ denote the set of codewords that encode the selected segments (we will prove that these codewords are fully computed at this point).
2. For all $t = 0, \ldots, \log(cN/\ell_{\mathsf{chunk}})$ let $I_t := \mathsf{HashMap}_{(cN/r),\epsilon}(2^t \cdot \ell_{\mathsf{chunk}}/r, \mathsf{rand}_1)$.
3. Consider the vector $\vec{w} \triangleq (w_1[I_{t_1}], \ldots, w_s[I_{t_s}])$, where each I_t is used for codewords of length $2^t \cdot \ell_{\mathsf{chunk}}/r$. Let $\sigma_x := \mathsf{LinHash}_\epsilon(\vec{w}, \mathsf{rand}_2)$ be a hash of the aforementioned vector. (rand_2 may be longer than required, in which case only use an appropriate prefix thereof.)
4. Return σ_x.

Figure 2: Subroutine for hashing.

1. Before starting the simulation, our simulator needs to execute the preprocessing phase for its components.
2. Our hashing process is more involved than that of [BK12], since we separate the input encoding phase from the codeword sampling phase. In addition, we apply a two level hash procedure.
3. We encode the "header" to the chunk using a separate error correcting code, and not as a part of the chunk. This is done in order to allow for the preprocessing to run in linear time rather than polynomial. See further discussion after Theorem 3.1 below.

Thread Thread_t

1. Consider a division of T into segments of length $2^t \cdot \ell_{\mathsf{chunk}}$, namely the k^{th} segment is $T[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} + 1 : (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$.
2. Encode each segment, in order, using the code $\mathcal{C}$. If the segment is not yet fully defined (namely T doesn't contain all the bits of that segment), then wait for it to be defined. If T rolls back, changing one of the segments, restart encoding that segment (once it is fully defined again).

Figure 3: Thread for encoding segments of length $2^t \cdot \ell_{\mathsf{chunk}}$.

3.1 Analysis

The following theorem summarizes the properties of $\mathsf{QLinSim}$:

Theorem 3.1. *For any protocol $\pi = (A, B)$ of communication complexity $N = \mathsf{CC}(\pi)$, the protocol $\mathsf{QLinSim}^{\pi}_{\epsilon=N^{-a}} = (\mathsf{QLinSim}^{A}, \mathsf{QLinSim}^{B})$ computes $\mathsf{Trans}(\pi)$, has communication complexity $\mathsf{CC}(\mathsf{QLinSim}^{\pi}) = O(N)$, and is robust with probability $\left(1 - 2^{-\Omega(1)\cdot aN}\right)$ to adversarial channels of constant ($\Omega(1)$) error rate. The computational complexity of $\mathsf{QLinSim}$ in the RAM model is at most $O(N \log N)$.*

Remark (Error Rate). In this work, we state robustness against an unspecified constant error rate, whereas some of the previous works are able to analyze for a specific constant (e.g. 1/32 for BK).

The reason is that we are compelled to encode the "headers" (the information the parties exchange to check the consistency of their local states) separately from the "payload" (the actual simulation of the next chunk). This is because we only allow linear time preprocessing for ExpSim, which corresponds to only being able to run it on short logarithmic chunks, too short to include the headers. We therefore use a separate error correcting code for the headers. On the other hand, BK combine the headers and the payload into one sequence, and uses ExpSim to exchange this entire sequence, which leads to optimal error rate.

The achievable error rate of our protocol is thus damaged, since the adversary can choose to corrupt either the header or the payload, and both options will "ruin" the current round for the communicating parties. The exact robustness parameter, therefore, is an intricate combination of the constants of the various components of our scheme, which we did not find very informative.

We do note, however, that given polynomial preprocessing time, we are able to run the preprocessing of ExpSim such that it will allow to bundle the header and payload as in BK. (The preprocessing is protocol independent and can be performed once and for all.) In such case, we can match the error rate of BK.

Theorem 3.1 is proven by combining Lemmas 3.4, 3.5 and 3.7 below.

We start by analyzing the segment selection process in the subroutine $\mathsf{TwoLevelHash}$. The next lemma shows that when a segment is "ready" according to the definition in the algorithm, then its encoding is complete.

Lemma 3.2. *If $i \geq (k+2) \cdot 2^t$ (i.e. $|T| \geq (k+2) \cdot 2^t \cdot \ell_{\mathsf{chunk}}$) then the encoding of the segment $[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} + 1, (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$ by the t^{th} thread is complete.*

Proof. Inductively on k: If $i \geq (k+2) \cdot 2^t$, then it means that at least 2^t rounds of the simulator have elapsed since the value of the transcript in our segment has been determined. Furthermore, by induction, the encoding of the previous segment already finished by the time our segment has been determined (the base case, $k = 0$ is obvious).

This means that at least $2^t \cdot O(\ell_{\mathsf{chunk}})$ computational steps have been devoted to the encoding of our segment. Since $\mathsf{FastEnc}$ runs in linear time, setting the constants properly will ensure that our segment finishes encoding on time. □

Next, we show that our greedy approach covers T by not-too-many segments.

Lemma 3.3. *The greedy algorithm in $\mathsf{TwoLevelHash}$ covers T by at most $O(\log N)$ segments.*

Proof. We prove by showing that at most two segments from each division (or thread) are used. We start by observing that the segments are chosen in decreasing length (= division) order: Consider a division t segment of the form $[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} : (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$, that has been chosen by the algorithm.

Now consider the algorithm's state at time point $(k-1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}$. If this time point occurs in the middle of a chosen segment, then this segment must be of division higher than t, and must be the one occurring right before the segment in discussion. If the algorithm is not in the middle of a segment, then it will necessarily choose a segment of division t or higher (since we know that the next segment from division t is ready). This establishes the claim.

Now, assume towards contradiction that there exists a division t from which the greedy algorithm uses 3 segments (or more). The first of these segments must start at an ending point of a segment of a higher division, namely a point in time of the form $k \cdot 2^{t+1} \cdot \ell_{\mathsf{chunk}}$. Since the 3rd segment in our sequence is ready, it means that $i \geq k \cdot 2^{t+1} + 4 \cdot 2^t = (k+2) \cdot 2^{t+1}$.

This is a contradiction since in that case, the greedy strategy would have favored a segment from the $(t+1)^{\text{th}}$ division instead of the first 2 segments from the t^{th} division. The result follows. □

We can now finally prove the running time of our simulator.

Lemma 3.4. *The computational complexity of our simulator is $O(N \log N)$.*

Proof. Preprocessing takes $O(N)$ time by Theorems 2.1, 2.5. This is followed by $O(N/\log N)$ rounds, in each of which the following is performed:

1. We run $O(\log N)$ computational steps in each of the $O(\log N)$ threads. This amounts to $O(\log^2 N)$ computational steps.
2. Sampling $\mathsf{rand}_{1,x}, \mathsf{rand}_{2,x}$ is done in $O(\log N)$ time.
3. The computational complexity of $\mathsf{TwoLevelHash}$ is $O(\log^2 N)$:
   (a) The greedy algorithm for finding the segment cover can be executed in logarithmic time (essentially it takes one pass over the binary representation of i).
   (b) Running $\mathsf{HashMap}$ for $O(\log N)$ times takes $O(\log^2 N)$ steps.
   (c) Computing $\vec{w}$ takes again $O(\log^2 N)$ time since there are $O(\log N)$ codewords, each accessed in $O(\log N)$ locations.
   (d) Running $\mathsf{LinHash}$ on $\vec{w}$ takes linear time in $|\vec{w}|$ (Corollary 2.4), namely $O(\log^2 N)$.
4. Encoding the "header" into ω_x using the linear code $\mathcal{C}$ takes $O(\log N)$ time.
5. Sending ω_x, receiving ω_y and decoding it takes $O(\log N)$ time (due to linear time decoding).
6. Another execution of $\mathsf{TwoLevelHash}$ takes $O(\log^2 N)$ time.
7. Running ExpSim to obtain L takes $O(\log N)$ time due to preprocessing (see Theorem 2.5).
8. The final decision and increment/decrement of T takes $O(\log N)$ time.

It follows that the total computational complexity per round is $O(\log^2 N)$. We conclude that the total computational complexity of $\mathsf{QLinSim}$ is

$$O(N) + O(N/\log N) \cdot O(\log^2 N) = O(N \log N) \; .$$ □

We proceed by analyzing the communication complexity of our simulator.

Lemma 3.5. *For any $\epsilon = 1/\mathrm{poly}(N)$, the communication complexity of $\mathsf{QLinSim}$ is at most $O(N)$.*

Proof. This follows in a straightforward manner since the protocol consists of $O(N/\log N)$ rounds and in each round the communication is $O(\log N)$. $\square$

Finally, we wish to prove the correctness of our protocol. We start by analyzing our two-phase hashing $\mathsf{TwoLevelHash}$.

Lemma 3.6. *Let $T_1 \neq T_2$ be two transcripts of the same length, and let $\mathsf{TwoLevelHash}^{T_i}(\cdot)$ denote the execution of $\mathsf{TwoLevelHash}$ w.r.t the transcript T_i. Then*

$$\Pr_{\mathsf{rand}_1,\mathsf{rand}_2}\left[\mathsf{TwoLevelHash}^{T_1}(\mathsf{rand}_1,\mathsf{rand}_2) = \mathsf{TwoLevelHash}^{T_2}(\mathsf{rand}_1,\mathsf{rand}_2)\right] \leq 2\epsilon \ .$$

Proof. If $T_1 \neq T_2$ then there must also be an inequality in one of the segments into which the transcripts are broken (recall that breaking into segments is only determined by the length of the transcript, not its content so both T_1, T_2 are broken in the same way). It follows that there exists some ℓ for which $w_\ell^{T_1}$ and $w_\ell^{T_2}$ have relative hamming distance at least δ. Theorem 2.3 guarantees, therefore, that

$$\Pr_{\mathsf{rand}_1}[\vec{w}^{T_1} = \vec{w}^{T_2}] \leq \epsilon \ .$$

Now, condition on the case that the values of $\vec{w}$ are different, we can use Corollary 2.4 which implies that

$$\Pr_{\mathsf{rand}_2}[\sigma_x^{T_1} = \sigma_x^{T_2}] \leq \epsilon \ .$$

Applying the union bound, the result follows. $\square$

Finally, we can prove the correctness of $\mathsf{QLinSim}$.

Lemma 3.7. *The simulator $\mathsf{QLinSim}$ is robust against to a constant fraction of adversarial errors, with failure probability at most $2^{-\Omega(\log_N(1/\epsilon)\cdot N)}$.*

This means that by choosing an appropriate inverse-polynomial ϵ, we can get the probability of error down to $2^{-\tau N}$ for any constant τ.

Proof. We follow the steps of [BK12, Lemma 3.4]. We define the following variables, which are, up to a name change, identical to those in [BK12]:

- Good transcript prefix (in chunks) good: This is the longest common prefix of the T value of the parties, rounded to whole chunks. Namely, if good' is the longest common prefix in bits, then $\mathsf{good} = \lfloor \mathsf{good}'/\ell_{\mathsf{chunk}} \rfloor$.
- Gap values $\mathsf{bad}_A, \mathsf{bad}_B$: We define $\mathsf{bad}_x \triangleq i_x - \mathsf{good}$, where i_A, i_B are the local values of i for the parties (naturally, bad_x is always non-negative).
- Error count e: This is the number of errors the adversary injected into the channel so far.
- Potential: We define a potential function

  $$\varphi \triangleq (\mathsf{good} - \mathsf{bad}_A - \mathsf{bad}_B)\cdot \ell_{\mathsf{chunk}} + \lambda_1 \cdot e \ ,$$

  where λ_1 is some constant to be defined later.

Our analysis will show that with all but $2^{-\Omega(\log_N(1/\epsilon)N)}$ probability, at the end of the execution it holds that $\varphi \geq (1+\lambda_2)N$. Letting E denote the total number of errors in the simulation, we get that if $E \leq \frac{\lambda_2}{\lambda_1}N$, then $\mathsf{good} \cdot \ell_{\mathsf{chunk}} \geq N$. The latter implies that the N-bit prefix of the local transcripts of both parties agree (and thus also agree with the transcript of π) and the simulation is a success. This implies robustness to error rate

$$\frac{(\lambda_2/\lambda_1)N}{\mathsf{CC}(\mathsf{QLinSim}^{\pi})} = \frac{\Omega(N)}{O(N)} = \Omega(1) ,$$

as required.

Let φ_ℓ be the change in the potential function in round ℓ of the protocol, namely

$$\varphi = \sum_{\ell=1}^{cN/\ell_{\mathsf{chunk}}} \varphi_\ell .$$

We will show that w.h.p, $\varphi_\ell \geq \ell_{\mathsf{chunk}}$ by case analysis:

- Consider a case where the adversary makes more than $\lambda_3 \ell_{\mathsf{chunk}}$ errors in round ℓ, where λ_3 is chosen so that $\lambda_3 \ell_{\mathsf{chunk}} = \min\{(\delta/2) \cdot |\omega_x|, \eta\ell_{\mathsf{chunk}}/\rho\}$ (note that this implies $\lambda_3 = \Theta(1)$).

  In such case, it holds that $\varphi_\ell \geq -O(1) \cdot \ell_{\mathsf{chunk}} + \lambda_1 \cdot \lambda_3 \cdot \ell_{\mathsf{chunk}}$, since e grows by $\lambda_3 \ell_{\mathsf{chunk}}$, and $\mathsf{good}, \mathsf{bad}$ can only change by a constant at each round. Therefore, choosing λ_1 big enough will imply the required outcome.

- If the adversary makes less than $\lambda_3 \ell_{\mathsf{chunk}}$ errors, then it means that both the decoding of ω_y by both parties and the simulation of L were successful. In this case, either one of the bad's shrinks, or if both are 0 then good increases. Therefore $\varphi_\ell \geq \ell_{\mathsf{chunk}}$ as required.

  The above, however, is only true so long as $\mathsf{TwoLevelHash}$ did not produce a collision between different transcripts. In such case, we might get $\varphi_\ell = -\lambda_4 \ell_{\mathsf{chunk}}$, where λ_4 is some constant (even in such problematic case, $\mathsf{good}, \mathsf{bad}$ cannot change by more than a constant).

  As in [BK12] we will use the independence of the hash seeds to bound the probability that above happens "too often". Using straightforward probability bounds (e.g. [BK12, Claim 2.1]), the probability that a false equality happens more than N/ℓ_{chunk} times is at most:

  $$2^{cN/\ell_{\mathsf{chunk}}} \cdot (4\epsilon)^{N/\ell_{\mathsf{chunk}}} .$$

  Namely, the probability of failure, for an inversely polynomial ϵ is at most

  $$2^{-\frac{\Omega(\log(1/\epsilon)) \cdot N}{\ell_{\mathsf{chunk}}} + O\left(\frac{N}{\log N}\right)} = 2^{-\Omega(\log_N(1/\epsilon)) \cdot N} .$$

The above implies that after cN/ℓ_{chunk} rounds, we get

$$\varphi \geq \left(\frac{cN}{\ell_{\mathsf{chunk}}} - \frac{N}{\ell_{\mathsf{chunk}}}\right) \cdot \ell_{\mathsf{chunk}} - \frac{N}{\ell_{\mathsf{chunk}}} \cdot \lambda_4 \ell_{\mathsf{chunk}} ,$$

that is

$$\varphi \geq (c - O(1)) \cdot N .$$

Selecting c to be a large enough constant will guarantee that $\varphi \geq 2N$ which implies correctness. □

4 $O(N)$-Time Simulator with Polynomially Small Error

In this section we present a different approach to interactive coding that allows to reduce the computational complexity to *linear* in the transcript length. However we will be forced to settle for only inverse polynomial success probability. For overview of our methods, see Section 1.2.

As in our previous algorithm (see Section 3), we consider a protocol π with communication complexity $\mathsf{CC}(\pi) = N$. The chunk length will be $\ell_{\mathsf{chunk}} \triangleq \lfloor \gamma \log N \rfloor$, where γ is as defined in Theorem 2.5.

Our simulator LinSim is defined in Figure 4. Similarly to the simulator $\mathsf{QLinSim}$ from Section 3, LinSim is parametrized by a constant c whose value will be determined in the analysis, where we assume w.l.o.g that cN/ℓ_{chunk} is an integer (not necessarily a power of 2 as before); and by a parameter $\epsilon = 1/\mathrm{poly}(N)$ that enables to control the failure probability of the simulator (the error probability will be roughly ϵN).

We make use of the inefficient simulator ExpSim (Theorem 2.5), the error correcting code $\mathcal{C}$ (Theorem 2.2) and the hash function $\mathsf{LinHash}$ (Corollary 2.4).

The parties will maintain the variable T of length at most cN that corresponds to its local view of the reconstructed transcript. However this value will only be used as input to the oracle machine A/B. The actual processing will be performed over the "hashed history" variable HH. Our simulator uses a stack as its main data structure.

4.1 Analysis

The following theorem summarizes the properties of LinSim.

Theorem 4.1. *For any protocol $\pi = (A, B)$ of communication complexity $N = \mathsf{CC}(\pi)$, the protocol $\mathsf{LinSim}^{\pi}_{\epsilon} = (\mathsf{LinSim}^{A}, \mathsf{LinSim}^{B})$ computes $\mathsf{Trans}(\pi)$, has communication complexity $\mathsf{CC}(\mathsf{LinSim}^{\pi}) = O(N)$, and is robust with probability $(1 - O(\epsilon N))$ to adversarial channels of constant ($\Omega(1)$) error rate. The computational complexity of LinSim in the RAM model is at most $O(N)$.*

Similarly to Theorem 3.1, we only state our result for an unspecified error bound, however we can match the error rate of BK if polynomial preprocessing is allowed. See the discussion following Theorem 3.1 for details.

Theorem 4.1 is proven by combining Lemmas 4.2, 4.3 and 4.8 below.

We start with the computational and communication complexities of LinSim, which are proven in a straightforward manner.

Lemma 4.2. *Let $\epsilon = 1/\mathrm{poly}(N)$. Then the running time of LinSim is at most $O(N)$.*

Proof. This follows directly from definition: The preprocessing of LinSim takes $O(N)$ steps. Then, in the main loop, we have $O(N/\log N)$ rounds and each requires $O(\log N)$ computation since $\mathsf{LinHash}$ runs in linear time and so does ExpSim given the preprocessing. □

Lemma 4.3. *Let $\epsilon = 1/\mathrm{poly}(N)$. Then the communication complexity of LinSim is at most $O(N)$.*

Proof. We have $O(N/\log N)$ communication rounds. Each round requires $O(\log N)$ communication for sending and receiving the headers (ω_x, ω_y), and additional $O(\log N)$ bits for the execution of ExpSim. □

We are left with proving robustness to a constant error fraction, which is a little more complicated. We start by formalizing our intuition from the overview above, that the adversary cannot cause the parties to agree on a false state by adding errors on the channel. Ultimately we want to show that

```
Simulator $\mathsf{LinSim}^X$

• Input: Oracle access to interactive machine $X$.
• Output: Transcript $T \in \{0,1\}^N$.
• Operation:
    0. Preprocessing: Run the preprocessing procedure $\mathsf{PreProc}_{\mathsf{ExpSim}}$ for the simulator $\mathsf{ExpSim}$, with $n = \ell_{\mathsf{chunk}} = \gamma \cdot \log N$ (see Theorem 2.5).
    1. Set $T := \phi$.
    2. Initialize a stack.
    3. Set $HH_x, HH_y := 0^{O(\log(1/\epsilon))}$, $L := 0^{\ell_{\mathsf{chunk}}}$, $\mathsf{rand}_x, \mathsf{rand}_y := 0^{q \log(O(\log(N/\epsilon))/\epsilon)}$, $i = 0$.
    4. Repeat $cN/\ell_{\mathsf{chunk}}$ times:
        (a) Sample $\mathsf{rand}'_x \leftarrow \{0,1\}^{q \log(O(\log(N/\epsilon))/\epsilon)}$.
        (b) Set $HH'_x := \mathsf{LinHash}_\epsilon((HH_x, HH_y, L, \mathsf{rand}_x, \mathsf{rand}_y, i), \mathsf{rand}'_x)$.
        (c) Encode $\omega_x := \mathsf{FastEnc}(HH'_x, \mathsf{rand}'_x, i)$.
        (d) Send $\omega_x$ over the channel and receive $\omega_y$.
        (e) Decode $\omega_y$ into $\mathsf{FastDec}(\omega_y) = (HH'_y, \mathsf{rand}'_y, j)$.
        (f) Compute $\sigma_y := \mathsf{LinHash}((HH_y, HH_x, L, \mathsf{rand}_y, \mathsf{rand}_x, j), \mathsf{rand}'_y)$. (Note the order change from the previous call to $\mathsf{LinHash}$ – this is since we are now recomputing a value of the other party.)
        (g) Use the simulator $\mathsf{ExpSim}$ to simulate $X$ for $\ell_{\mathsf{chunk}}$ more rounds. The output of $\mathsf{ExpSim}$ is a simulated transcript $L' \in \{0,1\}^{\ell_{\mathsf{chunk}}}$.
        (h) Proceed according to the following cases:
            • If $(i > j)$ then remove the last chunk from $T$. Pop stack and set values $(HH_x, HH_y, L, \mathsf{rand}_x, \mathsf{rand}_y, i)$ according to the popped entry (if the stack is empty, then set to initial values as in Step 3).
            • If $(i < j)$ then finish this iteration.
            • If $((i = j)$ and $(HH'_y \neq \sigma_y))$ then remove the last chunk from $T$. Pop stack and set values $(HH_x, HH_y, L, \mathsf{rand}_x, \mathsf{rand}_y, i)$ according to the popped entry (if the stack is empty, then set to initial values as in Step 3).
            • If $((i = j)$ and $(HH'_y = \sigma_y))$ then append the new chunk $L'$ onto $T$, set $i := i+1$, push $(HH_x, HH_y, L, \mathsf{rand}_x, \mathsf{rand}_y, i)$ into the stack, set $HH_x := HH'$, $HH_y := HH'_y$, $L := L'$, $\mathsf{rand}_x := \mathsf{rand}'_x$, $\mathsf{rand}_y := \mathsf{rand}'_y$.
    5. Output the first $N$ bits of $T$.
```

Figure 4: Interactive Simulator with Linear Computational Complexity.

disagreement can only be caused by "spontaneous" collisions in the hash functions, whose probability is bounded. We start by defining these collisions. We will use superscript A/B to indicate the local value of a variable inside A/B (respectively).

Definition 4.4. *Consider an execution of* $(\mathsf{LinSim}^A, \mathsf{LinSim}^B)$ *with an adversary that makes arbitrarily many errors. We say that there is a* collision *in round i of the protocol if either A or B samples a* rand *value that has been sampled before (by either of them); or if* $(HH_x^A, HH_y^A, L^A, \mathsf{rand}_x^A, \mathsf{rand}_y^A, i^A) \neq (HH_y^B, HH_x^B, L^B, \mathsf{rand}_y^B, \mathsf{rand}_x^B, i^B)$ *but these values collide upon application of* $\mathsf{LinHash}(\cdot, \mathsf{rand}'_x)$ *(of either party).*

We show that collisions only happen with probability $O(\epsilon N)$.

Lemma 4.5. *Consider an execution of* $(\mathsf{LinSim}^A, \mathsf{LinSim}^B)$ *with an adversary that makes arbitrarily many errors. The probability that a collision occurs during the execution of the protocol is at most* $O(\epsilon N)$.

Proof. The probability of the first cause of collision (sampling the same rand twice) is, by the union bound, at most

$$O(N/\log N) \cdot 2^{-q \log(O(\log(N/\epsilon))/\epsilon)} = O(N/\mathrm{polylog}(N)) \cdot \epsilon^q \leq O(\epsilon N) \ .$$

The probability for the second cause of collision is by the union bound and the properties of the hash function (see Corollary 2.4)

$$O(N/\log N) \cdot \epsilon = O(\epsilon N) \ .$$

Applying the union bound on the above implies the lemma. □

Next, we formalize what it means for the parties to "falsely agree" on a state: this is the case where the parties have a disagreement somewhere down the stack, but not in their top level variables.

Definition 4.6 (Synchronization)**.** *Consider an execution of* $(\mathsf{LinSim}^A, \mathsf{LinSim}^B)$ *with an adversary that makes arbitrarily many errors. If in the beginning of a round it holds that*

$$(HH_x^A, HH_y^A, L^A, \mathsf{rand}_x^A, \mathsf{rand}_y^A, i^A) = (HH_y^B, HH_x^B, L^B, \mathsf{rand}_y^B, \mathsf{rand}_x^B, i^B) \ , \tag{1}$$

then we say that the parties are locally synchronized*. Note that this implies that the parties have the same stack size i.*

If Eq. (1) *holds for every entry in the parties' stacks (and the stacks are of equal depth), then we say that they are* globally synchronized*. This implies, in particular, that $T^A = T^B$.*

We can finally prove that unless collisions occur, local synchronization implies global synchronization. That is, the parties cannot be made to falsely agree unless a collision occurred.

Lemma 4.7. *Consider an execution of* $(\mathsf{LinSim}^A, \mathsf{LinSim}^B)$ *with an adversary that makes arbitrarily many errors. If the parties are locally synchronized and there are no collisions, then they are also globally synchronized.*

Proof. Consider two parties that are locally synchronized. This means in particular that their stacks are of equal depth (since the value i corresponds to the depth of the stack).

Recall that HH_x is the hash value of the top entry in the stack using rand_x, and likewise for HH_y and rand_y. Since both parties agree on the HH and rand values, then they must also agree on the top entry in the stack. (Note that since the parties agree on $\mathsf{rand}_x, \mathsf{rand}_y$, it means that these values were not corrupted by the adversary, since each party knows for sure the value that he drew himself.)

The above argument can be extended inductively down the stack: If the parties agree on the entries at level i in the stack, then they must also agree on the entries in level $i-1$. Global synchronization follows. □

Finally we can prove the robustness of our simulator.

Lemma 4.8. *The simulator* LinSim *is robust against to a constant fraction of adversarial errors, with failure probability at most $O(\epsilon N)$.*

Given Lemma 4.7, the proof of this lemma is follows the same lines as Lemma 3.7 (in fact, it is simpler since the probability of collision is globally bounded). We repeat the proof for the sake of completeness.

Proof. We consider an execution of the protocol for which there are no collisions. By Lemma 4.5 this happens with probability $1 - O(\epsilon N)$ regardless of the adversary.

We define the following variables:

- Good stack prefix (in chunks) good: This is the longest prefix of the stack on which the parties agree.
- Gap values $\mathsf{bad}_A, \mathsf{bad}_B$: We define $\mathsf{bad}_x \triangleq i_x - \mathsf{good}$, where i_A, i_B are the local values of i for the parties (naturally, bad_x is always non-negative). This is the number
- Error count e: This is the number of errors the adversary injected into the channel so far.
- Potential: We define a potential function

  $$\varphi \triangleq (\mathsf{good} - \mathsf{bad}_A - \mathsf{bad}_B) \cdot \ell_{\mathsf{chunk}} + \lambda_1 \cdot e \ ,$$

  where λ_1 is some constant to be defined later.

We will show that at the end of the execution, it holds that $\varphi \geq (1+\lambda_2)N$. Letting E denote the total number of errors in the simulation, we get that if $E \leq \frac{\lambda_2}{\lambda_1}N$, then $\mathsf{good} \cdot \ell_{\mathsf{chunk}} \geq N$. The latter implies that the N-bit prefix of the local transcripts of both parties agree, since the local transcripts are identical to the concatenation of all the values L in the stack. Since these prefixes agree with each other, then they also agree with $\mathsf{Trans}(\pi)$ which means the simulation succeeded even with error rate

$$\frac{(\lambda_2/\lambda_1)N}{\mathsf{CC}(\mathsf{LinSim}^{\pi})} = \frac{\Omega(N)}{O(N)} = \Omega(1) \ ,$$

as required.

Let φ_ℓ be the change in the potential function in round ℓ of the protocol, namely

$$\varphi = \sum_{\ell=1}^{cN/\ell_{\mathsf{chunk}}} \varphi_\ell \ .$$

We will show that at every round $\varphi_\ell \geq \ell_{\mathsf{chunk}}$ by case analysis:

- Consider a case where the adversary makes more than $\lambda_3 \ell_{\mathsf{chunk}}$ errors in round ℓ, where λ_3 is chosen so that $\lambda_3 \ell_{\mathsf{chunk}} = \min\{(\delta/2) \cdot |\omega_x|, \eta \ell_{\mathsf{chunk}}/\rho\}$ (note that this implies $\lambda_3 = \Theta(1)$).

  In such case, it holds that $\varphi_\ell \geq -O(1) \cdot \ell_{\mathsf{chunk}} + \lambda_1 \cdot \lambda_3 \cdot \ell_{\mathsf{chunk}}$, since e grows by $\lambda_3 \ell_{\mathsf{chunk}}$, and $\mathsf{good}, \mathsf{bad}$ can only change by a constant at each round. Therefore, choosing λ_1 big enough will imply the required outcome.

- If the adversary makes less than $\lambda_3 \ell_{\mathsf{chunk}}$ errors, then it means that both the decoding of ω_y by both parties and the simulation of L' were successful.

  If the parties globally agree, then this means that $\mathsf{bad}_A = \mathsf{bad}_B = 0$ in which case both parties will push into the stack and good will increase by 1. If the parties globally disagree, then they also locally disagree by Lemma 4.7. This local disagreement will be detected and either one or both bad's will shrink by 1. Therefore $\varphi_\ell \geq \ell_{\mathsf{chunk}}$ as required.

The above implies that after cN/ℓ_{chunk} rounds, we get

$$\varphi \geq cN \ .$$

Selecting any $c > 1$ will guarantee that $\varphi \geq (1 + \Omega(1))N$ which implies correctness. □

References

[BEG+94] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1994.

[BK12] Zvika Brakerski and Yael Tauman Kalai. Efficient interactive coding against adversarial noise. In *FOCS*, pages 160–166. IEEE Computer Society, 2012.

[BR11] Mark Braverman and Anup Rao. Towards coding for maximum errors in interactive communication. In Lance Fortnow and Salil P. Vadhan, editors, *STOC*, pages 159–166. ACM, 2011.

[Bra12] Mark Braverman. Towards deterministic tree code constructions. In Shafi Goldwasser, editor, *ITCS*, pages 161–167. ACM, 2012.

[GI05] Venkatesan Guruswami and Piotr Indyk. Linear-time encodable/decodable codes with near-optimal rate. *IEEE Transactions on Information Theory*, 51(10):3393–3400, 2005.

[GMS11] Ran Gelles, Ankur Moitra, and Amit Sahai. Efficient and explicit coding for interactive communication. In Rafail Ostrovsky, editor, *FOCS*, pages 768–777. IEEE, 2011. Preliminary versions in [GS11, Moi11].

[Gol97] Oded Goldreich. A sample of samplers - a computational perspective on sampling (survey). *Electronic Colloquium on Computational Complexity (ECCC)*, 4(20), 1997.

[GS11] Ran Gelles and Amit Sahai. Potent tree codes and their applications: Coding for interactive communication, revisited. *CoRR*, abs/1104.0739, 2011.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[Jus72] J. Justesen. Class of constructive asymptotically good algebraic codes. *IEEE Trans. Inf. Theor.*, 18(5):652–656, September 1972.

[Moi11] Ankur Moitra. Efficiently coding for interactive communication. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:42, 2011.

[NN90] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. In Harriet Ortiz, editor, *STOC*, pages 213–223. ACM, 1990.

[RVW00] Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors. In *FOCS*, pages 3–13. IEEE Computer Society, 2000.

[Sch92] Leonard J. Schulman. Communication on noisy channels: A coding theorem for computation. In *FOCS*, pages 724–733. IEEE Computer Society, 1992.

[Sch93] Leonard J. Schulman. Deterministic coding for interactive communication. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *STOC*, pages 747–756. ACM, 1993.

[Sch96] Leonard J. Schulman. Coding for interactive communication. *IEEE Transactions on Information Theory*, 42(6):1745–1756, 1996. Journal version of [Sch92, Sch93] (refers mostly to the latter).

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell Systems Technical Journal*, 27:379–423, 623–656, 1948.

[Spi95] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. In Frank Thomson Leighton and Allan Borodin, editors, *STOC*, pages 388–397. ACM, 1995. Full version in [Spi96].

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de