Fast Interactive Coding Against Adversarial Noise*

Zvika Brakerski† Yael Tauman Kalai‡ Moni Naor§

August 7, 2014

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 arbitrarily, thus interrupting the execution of π (which was designed for an error-free 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) introduced the notion of *interactive coding*: A simulator that, given any protocol π, is able to simulate it (i.e. produce its intended transcript) even in the presence of constant rate adversarial channel errors, and with only constant (multiplicative) communication overhead. However, the running time of Schulman's simulator, and of all simulators that followed, has been exponential (or sub-exponential) in the communication complexity of π (which we denote by N).

In this work, we present three efficient simulators, all of which are randomized and have a certain failure probability (over the choice of coins). The first runs in time $\text{poly}(N)$, has failure probability roughly 2^{-N}, and is resilient to $\frac{1}{32}$-fraction of adversarial error. The second runs in time $O(N \log N)$, has failure probability roughly 2^{-N}, and is resilient to some constant fraction of adversarial error. The third runs in time $O(N)$, has failure probability $1/\text{poly}(N)$, and is resilient to some constant fraction of adversarial error. (Computational complexity is measured in the RAM model.) The first two simulators can be made *deterministic* if they are a priori given a random string (which may be known to the adversary ahead of time). In particular, the simulators can be made to be non-uniform and deterministic (with equivalent performance).

*This paper is the full version of "Efficient Interactive Coding Against Adversarial Noise" by Brakerski and Kalai (preliminary version FOCS 2012) and "Fast Algorithms for Interactive Coding" by Brakerski and Naor (preliminary version SODA 2013).

†Dept. of Computer Science and Applied Math, Weizmann Institute of Science. zvika.brakerski@weizmann.ac.il. Part of this work done while at Stanford University, supported by a Simons Postdoctoral Fellowship and by DARPA.

‡Microsoft Research, yael@microsoft.com.

§The Judith Kleeman Professorial Chair, Dept. of Computer Science and Applied Math, Weizmann Institute of Science, moni.naor@weizmann.ac.il. Research supported in part grants from the I-CORE Program of the Planning and Budgeting Committee, the Israel Science Foundation, BSF, IMOS and the Citi Foundation.

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 research area to this day. 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. One could consider error correcting against a *stochastic* channel that injects errors according to some distribution (such as the binary symmetric channel), but also against *worst case* channels that are allowed to inject errors adaptively and adversarially, so long as the prescribed error rate is not exceeded. This work will address the latter type of channels. A landmark in this area is the work of Justesen [Jus72] who showed the first explicit construction of such good codes. "Explicit" in this context refers to the ability to encode and decode in (uniform) polynomial time.

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 (e.g. [GI05]) improved the parameters of this result, achieving 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 codes 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, the communication complexity of S^π needs to be linearly related to that of the original π.

Schulman showed that interactive coding is achievable for an error rate of roughly $1/240$. However, the computational complexity of his simulator is $2^{\Theta(N)}$, where N is the length of the transcript of the protocol π. The high computational complexity stems from the use of a combinatorial object called a *tree code*, introduced in Schulman's work. Schulman showed how to construct and decode tree codes in exponential time, which implied the aforementioned exponential-time simulator S^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 $2^{\Theta(N^\epsilon)}$, respectively. Gelles, Moitra and Sahai [GMS11] showed that the computational complexity can be improved to $\text{poly}(N)$ 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.

1.1 Our Results

In this work, we construct three interactive coding schemes that are both efficient and resilient to adversarial errors, all three are over the binary alphabet. Our schemes are randomized, i.e. there is some probability of not succeeding in the simulation, and this probability is over the internal coin

flips of the two parties.[1] The first two schemes can be made deterministic in a non-uniform model of computation, as described below.

Polynomial-Time Simulator with Exponentially Small Failure Probability. Our first scheme PolySim (Section 3) consists of a simulator with computational complexity $\mathrm{poly}(N)$, failure probability at most $2^{-\mu N}$, where $\mu > 0$ can be any constant parameter, and is resilient to $1/32 - \epsilon$ error rate for any constant $\epsilon > 0$. The constants μ, ϵ determine the communication overhead (which is roughly $O(\mu)/\epsilon$). We further show that this performance is achievable even if the randomness is chosen ahead of time and not in the course of execution, and moreover there exists a random tape that is guaranteed to succeed for all protocols of certain transcript length. This immediately implies a non-uniform deterministic simulator by fixing the aforementioned random tape.

Theorem A. *For all constants $\mu, \epsilon > 0$, there exists an interactive simulator* PolySim *with communication complexity $O(N)$, failure probability $2^{-\mu N}$, and computational complexity* $\mathrm{poly}(N)$*, which is resilient to $1/32 - \epsilon$-adversarial error rate. The simulator* PolySim *can be made deterministic in a non-uniform model.*

More generally, we show how to take any inefficient interactive simulator running in time $2^{O(N)}$, and convert it into an efficient one. Our simulator is resilient to $\frac{(1-\epsilon)\cdot\eta}{4}$-fraction of adversarial error, where η is the error-rate of the inefficient interactive simulator. The best currently known asymptotic value of η is $1/8$ due to [BR11], which implies asymptotic rate of $1/32$ for our simulator. The factor $1/4$ loss in the error rate is a technical artifact of our analysis. While this is the best that we could extract from our methods, we do not see any barrier towards improving this factor.

Almost Linear-Time Simulator with Exponentially Small Failure Probability. Our second scheme QLinSim (Section 4) improves the computational complexity of the simulator. The performance here is comparable to PolySim, but the simulator runs in time $O(N \log N)$. The running time is measured in a model where the simulator has RAM access to its memory and oracle access to the machines which implement the original protocol. The tolerable error rate here, however, is only some unspecified constant. Error rate $1/32 - \epsilon$ as before can be achieved if we allow polynomial time (protocol and input independent) preprocessing. A deterministic non-uniform simulator is possible in this setting as well.

Theorem B. *For every constant $\mu > 0$, there exists an interactive simulator* QLinSim *with communication complexity $O(N)$, failure probability $2^{-\mu N}$, and computational complexity $O(N \log N)$, which is resilient to $\Omega(1)$-adversarial error rate (which can be improved to $1/32 - \epsilon$ if preprocessing is allowed). The simulator* QLinSim *can be made deterministic in a non-uniform model.*

Linear-Time Simulator with Polynomially Small Failure Probability. Lastly, our third scheme LinSim improves the running time to $O(N)$ (which is optimal in a model where the original protocol is provided as oracle). However, the cost here is that the failure probability drops to $1/N^{\mu}$, for any constant μ. The achievable error rate is similar to QLinSim.

Theorem C. *For every constant $\mu > 0$, there exists an interactive simulator* LinSim *with communication complexity $O(N)$, failure probability $N^{-\mu}$, and computational complexity $O(N)$, which is resilient to $\Omega(1)$-adversarial error rate (which can be improved to $1/32 - \epsilon$ if preprocessing is allowed).*

[1] The schemes are public-coin in the sense that we allow the adversary to know the internal state of the communication parties, including the random coins once they are flipped. We only require that the adversary does not know the value of the randomness before it is drawn.

1.2 Our Techniques

In what follows we present the high level overview of our constructions. We start by presenting our first simulator PolySim, and then explain how to improve the computational complexity to obtain QLinSim or LinSim.

PolySim: Polynomial-Time Simulator with Exponentially Small Failure Probability. As mentioned above, we build our first simulator by converting any exponential time simulator into an efficient one. Thus, our starting point is the aforementioned exponential time deterministic simulator from either [Sch96, BR11, Bra12], that we use in a black-box manner.[2] In order to use such a simulator and still achieve overall computational efficiency, we will use the simulator on logarithmic chunks of the protocol.

The basic idea is simple: Given a protocol $\pi = (A, B)$ with an N bit transcript, divide its transcript into $N/\log N$ chunks of $\log N$ bits each. Then run the exponential simulator chunk-by-chunk to reconstruct the entire transcript of π efficiently. This idea indeed seems to work in the stochastic model, where the errors are distributed roughly evenly between the different chunks (though the failure probability will not be negligible).

However, in the adversarial setting this idea is prone to failure, since we are only guaranteed that the *average* error rate over the chunks is constant, but it can be very high for any particular chunk. Namely, if the adversary introduces high noise rate at a specific chunk, then it can make the parties get that chunk all wrong, which will ruin correctness, even if all other chunks are computed correctly. We thus have to introduce some control mechanism by which the parties can identify that such erroneous event occurred, rewind their state back to the point of agreement, and try again. (This is the high level logic governing all known solutions starting from [Sch92].)

As a first solution, we introduce a synchronization check before each chunk, where the two parties compare their internal states to see that they are in sync. String comparison is known to be efficiently possible using (private, non-shared) randomness: each party will draw a randomness-efficient universal hash function, apply it to its local copy of the (simulated) transcript, and send the outcome, together with the description of the hash function, to the other party.[3] Using randomness efficient hash families [NN93, AGHP92], only $O(\log N)$ bits need to be sent to achieve good detection probability, and since the check only happens once for every $\log N$ bits chunk, its amortized effect on the information rate is constant. We note that 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.

This solution, however, does not work as is: The adversary might realize that the string comparison is the soft underbelly of our construction, and introduce errors during that stage. On the face of it, even a small amount of error can completely ruin the comparison.

One can overcome this problem in two ways: Either apply a standard (non-interactive) error correcting code to the (non-interactive) synchronization check, or incorporate the synchronization check as part of the chunk that is being communicated. We choose the latter since it allows us to prove higher error-resilience.[4] Namely, we consider a protocol that first sends (and receives) the synchronization information, and then the parties run the next chunk. We apply the exponential time

[2] This means that any future simulator, even non-tree based, can be used. In fact, we can also use probabilistic simulators with exponentially small error probability, but it would complicate the analysis somewhat.

[3] Interestingly, [Sch92] also uses universal hashing to compare the internal state, however he could not afford to send the description of the hash function along with the output, so he had to use pre-shared randomness.

[4] Jumping ahead, we note that our second and third simulator, whose aim is to further improve the computational complexity, use the former approach, and apply a (non-interactive) error correcting code to the synchronization check.

simulator to each extended chunk (synchronization + next chunk), whose communication complexity is still logarithmic. This will ensure that an adversary who wants to cause harm to the execution of any extended chunk needs to introduce at least $\Omega(\log N)$ errors (a constant fraction of the communication).

The parties will simulate each extended chunk over the channel and end up with a transcript, which contains the information of whether they are in sync or not, as well as the $\log N$ bits of transcript corresponding to the current chunk. If they were in sync, then they will use the $\log N$ bits of transcript, and continue to the next chunk. If they were not in sync, then they discard this information, and go back to the previous chunk (to try to get in sync).

Still there is a problem, since the adversary can adopt the following line of attack: It can invest enough errors to corrupt the view of *only one party*, and corrupt the check accordingly. In such a case, one party will revert to the previous chunk, while the other continues to the next. The protocol we described so far gives no mechanism to help the parties verify that they are computing the same chunk.

We therefore add an additional element to the synchronization check: in addition to the hash description and hash value, each party will also send its position in the simulated transcript, which comes at a tolerable cost of additional $O(\log N)$ bits (of course this is also incorporated into the protocol that is "protected" by the exponential simulator[5]). Given this information, it is possible to efficiently detect and correct gaps.

To analyze this protocol, we can think of the error correcting protocol as a game where we try to make the adversary waste its allotted number of errors, without setting the protocol back by too much. Intuitively, so long as the adversary only sets us back by less than the amount of steps required to recuperate (up to a constant), then our simulator will succeed. In our protocol, the adversary needs to invest $\Omega(\log N)$ errors to create an initial inconsistency, and it needs to keep investing $\Omega(\log N)$ errors in each following step to prevent our synchronization mechanism from recovering. Our analysis shows that the adversary will run out of errors at some point, allowing our recovery mechanisms to complete the simulation of the transcript of π.

The last risk that remains is that the adversary might corrupt the final chunks, leaving no time for recovery. This is treated similarly to previous works: We pretend that the transcript is actually longer than it really is by padding it with zeros. This way, a corruption at the end of the simulation can only harm the padding, which is thrown away anyway.

This simulator is formally presented and analyzed in Section 3.

A Deterministic Non-Uniform Simulator. The simulator PolySim (as well as QLinSim which is described below) can be made deterministic and always successful in a non-uniform model of computation where the simulator is allowed an $O(N)$ bit advice string *which is also known to the adversary.*

This is done, as is typical in applying the probabilistic method [AS92], by reducing the probability of the bad event (error) to be very small and then use a simple union bound. However, in our case we taking a union bound over all possible bad events requires some care, since the straightforward counting will fail to yield the desired result.

As described above, PolySim uses hash functions (chosen at random from an appropriate collection) in order to compare the states of the two communicating parties. This is the only place where randomness is used, and the only potential point of failure for the simulators. In fact, a careful examination of the analysis shows that the failure probability does not stem from the adversarial behavior, but rather from the probability that a constant fraction of the comparisons yield a false positive value. In particular, if this probability is 0, then the simulator will always succeed. If we use

[5]Jumping ahead, in the second and third simulator, this will be protected using a standard error-correcting code.

a hash family with seed length $\mu \log N$, then the probability of a false positive per comparison will be $N^{-\Omega(\mu)}$, and the total failure probability will decay as $2^{-\Omega(\mu)N}$. Increasing μ therefore reduces the failure probability, at the sole cost of increasing the communication complexity (though it remains $O(N)$ for any constant μ). In particular the number of rounds remains unchanged as μ increases.

We will next explain how taking μ to be a large enough constant implies that for all but an exponentially small fraction of possible random tapes, the probability of false positive occurring anywhere in the interaction is 0, regardless of the original protocol π and regardless of the adversarial behavior. This will immediately imply a non-uniform deterministic simulator by fixing the random tape of the simulator to one of those aforementioned good values.

if something is true on average with very high prob then it is true in the worst case, by a simple union bound. I would say: "Yes, it is true that in both the "standard" trick and in our case we apply a union bound over all possible "bad" events, and that in our case the counting of these "bad" events requires some care. The straightforward counting will fail to yield the desired result

We have to be careful in our counting, since a priori one may think that we need to counter *double* exponentially many possible protocols π (the number of different protocols of length N). However, as we shall see, this is not the case: let us count how many possible sequences of inputs to the hash function can occur when simulating a protocol with transcript length N. In each round of the simulated protocol, a hash function is applied to the internal state of each party. This internal state starts empty, and may change in poly(N) different ways at each round: Either an $O(\log N)$-bit chunk is appended, or a chunk is removed or there is no change. Therefore the total number of possible sequences of calls to the hash function is poly$(N)^r$, where $r = O(N/\log N)$ is the number of rounds, i.e. a total of $2^{O(N)}$ possible sequences. It is important to note that this number is independent of μ. For each such sequence of calls, the probability of failure of the simulator is $2^{-\Omega(\mu)N}$. It follows that taking μ to be a large enough constant, we get that the probability of failure at any possible execution also behaves as $2^{-\Omega(\mu)N}$ (regardless of the adversary's behavior!). The result follows.

For the formal analysis, see Section 3.3.

QLinSim: $O(N \log N)$-Time Simulator with Exponentially Small Failure Probability. The goal of QLinSim is to further improve the computational complexity of the first simulator from poly(N) to $O(N \cdot \log N)$.

The computational complexity of PolySim has two main sources: First, the exponential time simulator introduces some unspecified poly(N) computational complexity in each round of the protocol. Second, and perhaps more importantly, our first simulator 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 $\tilde{\Omega}(N^2)$.

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 (in the RAM model). Furthermore, this preprocessing does not depend on the specific protocol being simulated, only on its communication complexity.[6] 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

[6] As opposed to our first simulator, we don't know how to construct nearly linear time simulators based on any exponential time simulator. Our second and third simulators rely on the fact that the underlying exponential time simulator is one of the simulators in [Sch96, BR11], or any simulator that can be made to run in linear time given exponential-time preprocessing (and the preprocessing does not depend on the specific protocol being simulated, only on its communication complexity).

transcript of the simulation so far, and this decoding is independent of the specific protocol. Therefore, given that we apply the exponential simulator on chunks 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 (in the RAM model).

This solution, however, has an unfortunate implication: The header that contains the synchronization information cannot be made as short as we wish, and if the header is part of the chunk then its size is too large to allow linear time preprocessing. This is solved by encoding this header using a standard error correcting code with linear-time encoding and decoding. The outcome is that the tolerable error rate of QLinSim is an intricate function of the error rate and information rate of the error correcting code, the information rate, error rate and exponent of the exponential time simulator, and the properties of the hashing solution that we use (see below). We therefore do not provide a formula for the error rate, but rather show that it is some constant. As we mentioned above, allowing polynomial time preprocessing will allow to incorporate the header into the simulator as is done in our first simulator, and match its error rate.

We next turn to solve the second problem, which is the inefficiency caused by applying the hash function $\tilde{O}(N)$ times. 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, as in [NN93]. 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.[7] 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 the 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 a logarithmic scale), see Figure 1. 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 $O(\log^2 N)$ bits will still be too long to send over 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

[7] 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.

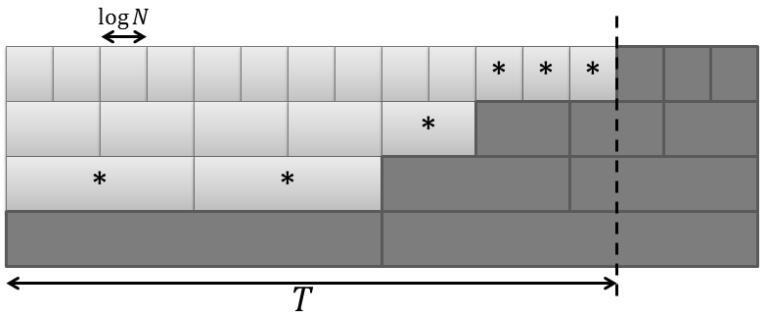

Figure 1: Dividing the transcript T into segments of all sizes. Each row represents a division. The light-colored segments are the ones that are "ready" at point T in time. The starred segments are the ones who will be chosen to "cover" the transcript T.

value with the desired properties.

Since the error resilience analysis of QLinSim is analogous to PolySim, we can apply the same strategy to obtain a deterministic simulator in the non-uniform setting with the same performance. We note that in the non-uniform setting, the preprocessing can also be considered as a part of the advice string (which will now be of size poly(N)) and an asymptotic error rate of $1/32$ is achievable.

The simulator QLinSim is formally presented and analyzed in Section 4.

LinSim: $O(N)$-Time Simulator with Polynomially Small Failure Probability. 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-time algorithm will use this hash value as 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 to $T\|L$ 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 hash into the same value) even at a single 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. Even if the hash function is chosen in a way that the parties' transcripts are in distinct equivalence classes, the adversary might cause the parties to receive the wrong function, so that they are led to believe that they are in agreement. 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 the analysis becomes very similar to the analysis of the first simulator.

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 this work. One important idea is using a succinct authentication protocol in order to check the previous transcript: that it in order to authenticate a large message it is possible to reduce it to authenticating a much shorter message (see [GN93, NSS08] where the technique of cooperative hashing is used). Other inspirations come from the work on *incremental cryptography* (see Bellare et al. [BGG94]) and memory checking (see Blum et al. [BEG+94]) where the goal is to perform some sort of authentication of the current state of the data structure *without rereading the full memory*. The tricky part is where should we embed the secret key (= hashing seed in our context), since in our setting there are no shared secrets or a shared trusted key. One way to view our solution to this problem is that the key is generated by one of the parties and sent to the other party over the channel, however the transmission is postponed until *after* the key has been used. Namely, we send our hash function to the other party only after it had been applied to the (past) transcript. This is somewhat reminiscent of the way authentication is achieved in the Tesla protocol of Perrig et al. [PCTS00] which is meant for synchronized environments. This postponement raises its own issues that we need to deal with, most obviously the adversary's ability to corrupt both the hash function and value at the same time.

1.4 Followup Work and Open Problems

The main open problem this work suggests is the explicit construction of a deterministic algorithm for interactive coding with polynomial or near linear complexity (whose existence is assured by Theorem 3.7). An additional obvious challenge is to characterize the maximal tolerable error rate by interactive coding schemes, and more generally the trade-off between communication overhead and tolerable error. Braverman and Rao [BR11] describe an upper bound of $1/4$ on the tolerable error rate in a model where each party is required to speak "in turn". Ghaffari, Haeupler and Sudan [GHS13] suggest a model where time slots are not assigned to parties, and suggest that the upper bound in this setting may be $2/7$. They were able to match this bound using a scheme with communication complexity N^2. This was later improved to slightly super-linear communication complexity by Ghaffari and Haeupler [GH13], using an explicit protocol. An alternative model with slightly different bounds was suggested by Agrawal, Gelles and Sahai [AGS13]. However, the problem is still open when communication complexity is bound to be linear.

Studying the maximal tolerable error rate with linear communication is only a special case of the more general challenge which is to characterize the trade-off between tolerable error rate and communication overhead. A first step in this direction was taken by Kol and Raz [KR13] who studied the interactive channel capacity of the binary symmetric channel with asymptotically small noise rate. In other words, their work studies the error-communication trade-off for such channels. An obvious open problem is to extend their findings to worst-case channels and additional error regimes.

In the aforementioned works of [GHS13, AGS13], the parties do not speak "in turn", but time is still divided into well defined time slots. A yet unexplored area is that of protocols that work in a completely asynchronous environment, where it is not obvious how to map the bits sent to the bits received.

Lastly, the question of whether interactive coding conflicts with input privacy has been explored by Chung, Pass and Telang [CPT13], and by Gelles, Sahai and Wadia [GSW14]. These works suggest that the "rewinding" process that is implicit in all interactive coding schemes is in fact necessary, and it may allow the communicating parties (or the adversary) to learn more about the inputs of the parties than what it could have learned in the noiseless setting.

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. The most straightforward model is to consider the simulator as an interactive Turing machine with oracle access to another interactive machine X (representing the party being simulated). An alternative, polynomially related, model is a RAM model where the simulator is a RAM machine with a logarithmic (in the communication complexity) word length, and with oracle access to the interactive machine X. In the RAM model, the communication with the oracle X is by using a designated area in the memory. Writing T in this designated area and making an oracle call, will write the next message $X(T)$ into a (different) area in the memory. We 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. The complexity in the RAM model is the total number of memory read/writes and oracle calls.

In the first result in this paper (the simulator $\mathsf{PolySim}$) we will not care about polynomial slowdown in the communication complexity and therefore the simulator will be presented in the simpler Turing machine model. When we attempt to optimize the computational complexity (in our simulators $\mathsf{QLinSim}$ and LinSim) we will use the RAM model which will allow us to analyze the complexity more accurately.[8]

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.

[8] Note that in the Turing machine model, even simulation of the protocol *without channel noise at all* will incur a complexity cost of N^2, since in each round, the simulator needs to feed the transcript so far into its oracle, which takes linear time in the transcript length.

2.3 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 2.1 (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 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 until this point in time, 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. □

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

3 Poly-Time Simulation with Exponential Failure Decay

In this section, we show how to convert any interactive protocol $\pi = (A, B)$ into one that is resilient to constant fraction of adversarial errors, and is efficient in the sense that it computes $\mathsf{Trans}(\pi)$ with a polynomial overhead in the time complexity. Recall that without loss of generality A, B are deterministic and don't take any input.

More specifically we present an efficient simulator $\mathsf{PolySim}$ that has oracle access to either A or B, and simulates these parties in an error resilient manner, so that the new protocol $(\mathsf{PolySim}^{A}, \mathsf{PolySim}^{B})$ computes $\mathsf{Trans}(\pi)$, even in the presence of constant fraction of adversarial errors. We often use X to indicate one of $\{A, B\}$, in which case Y will denote the other party.

Our simulator uses a randomness-efficient string comparison test. Such is provided by using the families of hash functions of either [NN93, AGHP92] (see in particular [NN93, Section 9]). In what follows, we denote $\{0,1\}^{\leq n} \triangleq \cup_{i \in [n]} \{0,1\}^{i}$.

Theorem 3.1 ([NN93, AGHP92])**.** *There exists a constant $q > 0$ and an ensemble of hash families $\{\mathcal{H}_k\}_{k \in \mathbb{N}}$ such that for every $k \in \mathbb{N}$ and for every $h \in \mathcal{H}_k$, $h : \{0,1\}^{\leq 2^k} \to \{0,1\}^{q \cdot k}$ is poly-time*

computable, it is efficient to sample $h \leftarrow \mathcal{H}_k$ using only $q \cdot k$ random bits, and for all $x \neq y \in \{0,1\}^{\leq 2^k}$ it holds that

$$\Pr_{h \leftarrow \mathcal{H}_k}[h(x) = h(y)] \leq 2^{-k} .$$

Note that we assume w.l.o.g that the seed length and output size of the hash function are identical (otherwise, define q according to the maximal of the two).

The simulator PolySim is presented in Section 3.1 and analyzed in Section 3.2.

3.1 Simulator $\mathsf{PolySim}^{\pi}$

Let N be an upper bound on the communication complexity of $\pi = (A, B)$, such that N is a power of 2. Throughout the simulation, party X maintains a local variable T_X that represents its current view on the partial transcript of π. At the end of the algorithm, the N-prefix of T_X will be equal to $\mathsf{Trans}(\pi)$.

We consider the simulator ExpSim from Theorem 2.1 with parameters ρ, η,[9] and the hash family from Theorem 3.1 with parameter q. We let μ be a parameter that will determine the exponent of the failure probability. We define a parameter τ which will be useful in the presentation of the algorithm and in the analysis. Intuitively, τ represents the communication blowup incurred by each "round" of the simulation (see below).

$$\tau \triangleq \frac{4\mu q + 3}{\rho} . \tag{1}$$

As outlined in Section 1.2, our simulator $\mathsf{PolySim}^{\pi} = (\mathsf{PolySim}^{A}, \mathsf{PolySim}^{B})$ (Figure 2) works in $O(N/\log N)$ rounds. Each round contains an execution of a logarithmic-communication subroutine Chunk (Figure 3), which is "protected" from channel errors using the exponential time simulator ExpSim. (Of course this protection can sometimes fail when there are too many errors, but we will show that the process converges nonetheless.) In total, each round communicates $\tau \log N$ bits. We elaborate more on the subroutine and the use of ExpSim, below.

The subroutine Chunk communicates the synchronization information, as well as (what it believes to be) the next chunk of the transcript between the two parties. This subroutine is never "really" executed, but rather simulated by the exponential time simulator ExpSim (Chunk only communicates $O(\log N)$ bits). The simulator ExpSim returns a "protected transcript" of the execution of Chunk. This transcript may be that of a legal execution if there were not too many errors, or it can be completely arbitrary if there were.

Given the protected transcript, the parties can check if their states are in sync. If not, they can move towards rectifying the situation. If they were in sync, then the protected transcript indeed contains the next chunk of $\mathsf{Trans}(\pi)$, which can be appended to the local copy of the transcript. Of course there is always the chance that the protected transcript is wrong, but our analysis shows that this does not set us back by too much.

Lastly, a parameter c controls the number of rounds of our simulator. The value of c determines the convergence of PolySim to its asymptotic tolerable error rate. The larger c is, the closer the simulator gets to tolerating $\eta/4$ fraction of errors. We will assume w.l.o.g that $cN/\log N$ is integer.

Since the typical value for c is a large constant (in fact, Theorem 3.2 is meaningless unless $c > 5$), the local transcripts that the parties maintain are longer than N. The output of the simulator will be

[9] In fact we can use any simulator that has $2^{O(N)}$ computational overhead (and is resilient to a constant fraction of adversarial error and has a constant blowup in communication).

Simulator $\mathsf{PolySim}^X$

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

1. Set $T := \phi$.
2. Repeat $cN/\log N$ times:
   (a) Sample a new hash function $h_x \leftarrow \mathcal{H}_{\mu\log(N)}$ (recall Theorem 3.1). Set $\sigma_x := h_x(T)$.
   (b) Let X' be shorthand notation for the subroutine Chunk with the current variable values (T, h_x, σ_x). Formally: $X' \triangleq \mathsf{Chunk}^X_{T,h_x,\sigma_x}$.
   (c) Use the simulator $\mathsf{ExpSim}^{X'}$ to simulate X'. The output of ExpSim is a simulated transcript of the form: $(i, \tilde{h}_x, \tilde{\sigma}_x) \| (j, h_y, \sigma_y) \| L$, where $|L| = \log N$.
   (d) If $(i \neq |T| / \log N)$ or $(\tilde{h}_x \neq h_x)$ or $(\tilde{\sigma}_x \neq \sigma_x)$, then finish this iteration.
   Otherwise we proceed with one of the following cases:
   - If $(i > j)$ then set $T := T_{\leq (i-1)\log N}$.
   - If $(i < j)$ then finish this iteration.
   - If $((i = j)$ and $(h_y(T) \neq \sigma_y))$ then set $T := T_{\leq (i-1)\log N}$.
   - If $((i = j)$ and $(h_y(T) = \sigma_y))$ then set $T := T \| L$.
3. Output $T_{\leq N}$.

Figure 2: Our simulator.

the N-bit prefix of this long local transcript. Therefore, if the adversary corrupts the last round of the execution, it will not affect the output, which is the N-bit prefix of the local transcripts.

The simulator is presented in Figure 2. The subroutine Chunk is presented in Figure 3.

3.2 Analysis

The following theorem summarizes the properties of our simulator.

Theorem 3.2. *For any protocol $\pi = (A, B)$ of communication complexity $\mathsf{CC}(\pi) = N$, the protocol $\mathsf{PolySim}^\pi = (\mathsf{PolySim}^A, \mathsf{PolySim}^B)$ computes $\mathsf{Trans}(\pi)$, has communication complexity $\mathsf{CC}(\mathsf{PolySim}^\pi) = O(\mu c) \cdot N = O(N)$, and is robust with probability $\left(1 - 2^{-(\mu - o(1))N}\right)$ to adversarial channels of error rate $(1 - 5/c) \cdot (\eta/4)$. The computational complexity of $\mathsf{PolySim}$ is at most $\mathrm{poly}(N)$.*

The theorem follows by combining Lemmas 3.3, 3.4, and Corollary 3.6 of Lemma 3.5, below. In what follows, recall that the constant q is from Theorem 3.1, the constants ρ and η are from Theorem 2.1, and the constant τ is defined in Eq. (1).

Lemma 3.3. *It holds that*

$$\mathsf{CC}(\mathsf{PolySim}^\pi) \leq c\tau N = O(\mathsf{CC}(\pi)) \ .$$

Subroutine $\mathsf{Chunk}^X_{T,h_x,\sigma_x}$

i. Let $i := |T| / \log N$, represented as a bit string of length $\log N$.

ii. Send (i, h_x, σ_x) over the channel and receive (j, h_y, σ_y). (This is of course done bit by bit.)

iii. Execute $X(T)$ for $\log N$ communication steps.

Figure 3: Subroutine to be fed into ExpSim.

Proof. The protocol $\mathsf{PolySim}^\pi$ is composed of $c \cdot N/\log N$ rounds, each containing a simulation of X' which communicates $\frac{2(2\mu q+1)\log N+\log N}{\rho} = \tau \log N$ bits. The communication complexity is therefore:

$$\mathsf{CC}(\mathsf{PolySim}^\pi) = (cN/\log N) \cdot (\tau \log N) = c\tau N = O(N) \ . \qquad \square$$

Lemma 3.4. *The computational complexity of* $\mathsf{PolySim}^\pi$ *is at most* $\mathrm{poly}(\mathsf{CC}(\pi))$.

Proof. The simulator $\mathsf{PolySim}$ runs in $c \cdot N/\log N = O(N/\log N)$ rounds. In each round, the simulator ExpSim is called on a machine X' that communicates $2(2\mu q+1)\log N + \log N = (4\mu q+3)\log N$ bits. The computational complexity of ExpSim on such machines is $\mathrm{poly}(N)$. In addition, each party makes two evaluations of hash functions, which contribute additional $\mathrm{poly}(N)$ computational steps. All of the other operations are simple manipulations on the transcript. $\square$

We can make the analysis above more specific: Letting t_h denote the time complexity of the function family $\mathcal{H}_{\mu \log(N)}$; and letting t denote the time complexity of ExpSim when executed on protocols of communication complexity $(4\mu q + 3)\log N$, we get that the computational complexity of $\mathsf{PolySim}^\pi$ is at most $O((N/\log N) \cdot (t_h + t + \log N))$ (in some reasonable computational model). Using known instantiations, we can get $t_h = \Theta(N)$ (note that the hash function must read all of its input). We are not aware of a precise analysis as to the running time of ExpSim, so we can only say that $t = \mathrm{poly}(N)$ for an unspecified polynomial.

The following lemma proves the success probability of our simulation over adversarial noisy channels. This is the heart of our analysis which is used to derive Corollary 3.6 below.

Lemma 3.5. *When running* $\mathsf{PolySim}^\pi$ *over a channel that makes at most*

$$E = (c-5) \cdot \frac{\eta\tau}{4} \cdot N$$

adversarial errors, $\mathsf{PolySim}^\pi$ *outputs* $\mathsf{Trans}(\pi)$ *with probability at least* $1 - 2^{-(\mu - o(1))N}$.

Proof. Let $\pi = (A, B)$ and consider an execution of $\mathsf{PolySim}^\pi = (\mathsf{PolySim}^A, \mathsf{PolySim}^B)$. For $X \in \{A, B\}$, we denote the values computed by $\mathsf{PolySim}^X$ during the protocol with subscript X (e.g., T_A or i_B). We denote $i_X \triangleq |T_X| / \log N$ (note that this is always an integer).

We define the following (random) variables:

- Good transcript prefix (in chunks) g: This is the longest common prefix of T_A, T_B, rounded to whole chunks. Namely, if g' is the longest common prefix in bits, then $g = \lfloor g'/\log N \rfloor$.
- Gap values α_A, α_B: We define $\alpha_X \triangleq i_X - g$ (naturally, α_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

$$\varphi \triangleq (g - \alpha_A - \alpha_B) \cdot \log N + \frac{4}{\eta\tau} \cdot e \ ,$$

  where τ is as defined in Eq. (1).

We show that when the algorithm terminates, it holds, with probability $1 - 2^{-(1-o(1))N}$, that

$$\varphi \geq (c-4)N \ .$$

This implies that

$$g \log N \geq \varphi - \frac{4}{\eta\tau} \cdot E \geq (c-4)N - (c-5)N = N \ ,$$

which in turn means that the two parties agree on the prefixes $T_{A,\leq N} = T_{B,\leq N} = \mathsf{Trans}(\pi)$.

We remark that the coefficient 4 in the definition of the potential function is "responsible" for the loss in the error rate of our simulator ($\eta/4$ compared to η). In the course of presenting our analysis, we will explain what warrants this factor.

Our proof follows by showing that the potential function φ must grow by roughly $\log N$ with every round of the protocol. Let φ_ℓ denote the change in φ in iteration ℓ (where $\ell = 1, \ldots, cN/\log N$). We use case analysis to lower-bound φ_ℓ:

We will use the following notation. For all ℓ we denote by F_ℓ the event where $i_A = i_B$, $T_A \neq T_B$ and yet either $h_A(T_A) = h_A(T_B)$ or $h_B(T_A) = h_B(T_B)$. Note that by Theorem 3.1, $\Pr[F_\ell | F_1, \ldots, F_{\ell-1}] \leq 2 \cdot N^{-\mu}$, since h_A, h_B are sampled independently of the history.

- **Case 1:** The number of errors in the iteration is at most $\eta\tau \log N$. In this case we are guaranteed that Q simulates A', B' correctly (where A' is the machine X' defined by $\mathsf{PolySim}^A$ in Step 2b, and B' is the same for $\mathsf{PolySim}^B$). Therefore the output of Q (for both parties) is the real transcript of the protocol (A', B').

  Again we have a few cases.

  - If $i_A \neq i_B$, then both parties will get $i \neq j$. In such case, it must be that for some X, $\alpha_X > \alpha_Y \geq 0$. The larger α_X necessarily belongs to the party with the larger i_X, and this party will chop a chunk off its transcript. We conclude that $\varphi_\ell \geq \log N$.
  - If $i_A = i_B$ and $T_A = T_B$, then this means that both partial transcripts agree. In this case, L is indeed the next chunk of the execution and both parties will append it to their transcripts. We conclude that $\varphi_\ell \geq \log N$.
  - If $i_A = i_B$ and $T_A \neq T_B$, then the outcome depends on the event F_ℓ defined above. If F_ℓ does not happen, then both α_A, α_B decrease by 1 and $\varphi_\ell \geq 2 \log N$. If F_ℓ does happen, then one or two of the parties might append a faulty L, causing α_A, α_B to increase by 1, namely $\varphi_\ell \geq -2 \log N$. We conclude that $\varphi_\ell \geq 2 \log N (1 - 2 \cdot \mathbb{1}_{F_\ell})$.

- **Case 2:** The number of errors in the iteration is greater than $\eta\tau \log N$. In this case, all bets are off and the expression $(g - \alpha_A - \alpha_B)$ can decrease by at most 3 (the worst case is when g decreases by 1 and i_X increases by 1, causing α_X to increase by 2, note that $\alpha_Y = 0$ in this case). We conclude that

$$\varphi_\ell \geq -3 \log N + \frac{4}{\eta\tau} \cdot (\eta\tau) \log N \geq \log N \ .$$

The above equation explains the need for a factor 4 in the definition of the potential function, which is responsible for the loss in error rate. (One could think that any factor larger than 3 should be sufficient, but we found it problematic for other parts of the analysis.)

We conclude that either $\varphi_\ell \geq \log N$ or $\varphi_\ell \geq 2\log N(1 - 2 \cdot \mathbb{1}_{F_\ell})$. Let K be the set of rounds for which the latter holds, and let $k = |K|$. Then

$$\begin{aligned}
\varphi &= \sum_{\ell=1}^{cN/\log N} \varphi_\ell \\
&\geq (cN/\log N - k)\log N + \sum_{\ell \in K} 2\log N(1 - 2 \cdot \mathbb{1}_{F_\ell}) \\
&= cN + k\log N - 4\log N \sum_{\ell \in K} \mathbb{1}_{F_\ell} \\
&\geq cN + k\log N - 4\log N \sum_{\ell=1}^{cN/\log N} \mathbb{1}_{F_\ell} \ .
\end{aligned}$$

To bound the latter expression, we recall that $\Pr[F_\ell | F_1, \ldots, F_{\ell-1}] \leq 2 \cdot N^{-\mu}$, and therefore

$$\Pr\Big[\underbrace{\sum_{\ell=1}^{cN/\log N} \mathbb{1}_{F_\ell} \geq N/\log N}_{\text{Denote this event by } F}\Big] \leq 2^{cN/\log N} \cdot (2 \cdot N^{-\mu})^{N/\log N} = 2^{-(\mu - o(1))N} \ . \tag{2}$$

If the above bad event does not happen (i.e., if $\sum_\ell \mathbb{1}_{F_\ell} < N/\log N$), then

$$\varphi \geq cN + k\log N - 4N \geq (c-4)N \ ,$$

and the lemma follows. □

Finally, the error rate for which robustness holds follows immediately.

Corollary 3.6. $\mathsf{PolySim}^\pi$ *is robust with probability* $\big(1 - 2^{-(\mu - o(1))N}\big)$ *to adversarial channels of rate*

$$\big(1 - 5/c\big) \cdot \frac{\eta}{4} \ .$$

Proof. Combining Lemma 3.3 and Lemma 3.5, it follows that with probability $1 - 2^{-(\mu - o(1))N}$, the protocol $\mathsf{PolySim}^\pi$ is robust to noise rate

$$\frac{E}{\mathsf{CC}(\mathsf{PolySim}^\pi)} = \frac{(c-5)\eta\tau N}{4 \cdot c\tau N} = (1 - 5/c) \cdot \frac{\eta}{4} \ .$$ □

3.3 A Deterministic Non-Uniform Solution

We now show that using a proper error analysis, we can change the order of quantifiers in Theorem 3.2 and show that with high probability over the random tape of the simulator, it will be successful against *any* adversary (with only minimal loss in the failure probability). The implication is that a deterministic non-uniform solution follows by fixing such a good random tape as an advice string. A formal statement and proof follows.

Theorem 3.7. *Consider the simulator* PolySim *as stated in Theorem 3.2. With probability* $(1 - 2^{-(\mu - O(c))})$ *over the random tape of* PolySim*, the simulator succeeds in simulating* any *protocol of communication complexity* N *against* any *adversarial channel (of error rate* $(1 - 5/c) \cdot (\eta/4)$*, as in Theorem 3.2).*

Since we can grow μ independently of c, we can choose the parameters so that "bad" random tapes are exponentially improbable.

Corollary 3.8. *There exists a deterministic (and thus failure free) simulator in a non-uniform model that achieves the same performance as* PolySim*.*

Proof of Theorem 3.7. The theorem will follow by a union bound over all possible internal states of the communicating parties throughout the protocol. Consider an execution of PolySim and let $\mathcal{T}$ denote the sequence of pairs of internal states (T_A, T_B) as recorded in each round of the simulation (a total of $cN/\log N$ pairs).

A simple counting argument shows that there are at most $2^{O(c)N}$ possible values for $\mathcal{T}$. To see this, notice that in each round of the protocol, the internal state can either decrease by one chunk, remain unchanged, or increase by one chunk (an added chunk contains $\log N$ bits, so it can take $2^{\log N}$ possible values). Therefore the internal transcript of each party can change in one of $1 + 1 + 2^{\log N} = (N + 2)$ many ways, and the transcript of both parties can change in one of at most $(N + 2)^2$ many ways in each round. Therefore, if we count over all $cN/\log(N)$ rounds, we get that there are at most $(N + 2)^{2cN/\log(N)} = 2^{O(c)N}$ possible values for $\mathcal{T}$. We will use this bound in our argument below.

We now recall the proof of Lemma 3.5, and notice that failure can occur only if the event F from Eq. (2) occurs. Furthermore, F is fully determined by the value for $\mathcal{T}$ and the random tape of the simulator (i.e. the random tape of both simulating parties), which we denote here by r. We can therefore think of F as a deterministic event $F(\mathcal{T}, r)$, which counts the number of collisions that the hash functions specified by r induce on the sequence $\mathcal{T}$, and event $F(\mathcal{T}, r)$ occurs only if this number is at least $N/\log(N)$ (as specified in Eq. (2)).

Consider a value of r for which for every possible $\mathcal{T}$ it holds that $\mathbb{1}_F(\mathcal{T}, r) = 0$ (if such exists). Then in particular, such values of r make the simulator succeed regardless of the adversarial behavior, since whatever the adversary does, it cannot steer the simulated protocol to a state where F occurs. We will show next that such values of r are abundant.

Let us start by fixing a value for $\mathcal{T}$ and sampling a value for r uniformly at random.[10] In such case, the bound from Eq. (2) still holds, and we get that

$$\Pr_{r \in_R U}[F(\mathcal{T}, r)] \leq 2^{-(\mu - o(1))N}.$$

It thus follows by the union bound that

$$\Pr_{r \in_R U}[\exists \mathcal{T} \text{ s.t. } F(\mathcal{T}, r)] \leq 2^{O(c)N} \cdot 2^{-(\mu - o(1))N} = 2^{-(\mu - O(c))N}.$$

Our conclusion is that for all but $2^{-(\mu - O(c))N}$ fraction of the values of r, it indeed holds that for every possible $\mathcal{T}$ indeed $\mathbb{1}_F(\mathcal{T}, r) = 0$, as desired. □

[10] Note that we *do not* claim that the actual distribution of r conditioned on $\mathcal{T}$ in any execution of the protocol is uniform. The distribution here is just an analytical tool for understanding the behavior of $F(\mathcal{T}, r)$.

4 Almost Linear Time Simulator with Exponential Failure Decay

In this section we improve the computational overhead of the simulator presented in Section 3. Specifically, the oracle-RAM complexity of our new simulator is $O(N \log N)$, compared to the trivial lower bound of $\Omega(N)$. Similarly to the simulator presented in Section 3, our new simulator is randomized and succeeds with probability $2^{-\mu N}$ where μ can be *any* constant. Namely, we can reduce the error term as much as we wish.

Recall that the computational complexity of our first simulator (presented in Section 3) comes from two main ingredients: First, the underlying exponential time simulator introduces some unspecified $\text{poly}(N)$ computational complexity in each round of the protocol. Second, and perhaps more importantly, our first simulator 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 $\tilde{\Omega}(N^2)$.

We deal with the first problem by using one of the exponential time simulators from [Sch96, BR11]. These simulators have the property that they can be made to run in *linear* time, given exponential-time preprocessing (in the RAM model), and this preprocessing does not depend on the specific protocol being simulated, only on its communication complexity. Therefore, if we choose our chunk size to be small enough, i.e. some $\gamma \log N$ for a small constant γ, then 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 $O(N)$.

This solution, however, has an unfortunate implication: The header that contains the synchronization information cannot be made as short as we wish, and if the header is part of the chunk then its size is too large to allow linear time preprocessing. This is solved by encoding this header using a standard error correcting code with linear-time encoding and decoding.

Theorem 4.1 ([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.*

To solve the second problem we need to boost the efficiency of the hashing process. To this end, we divide the hashing process into three different parts with fairly weak dependence: The hashing process starts by encoding the input with a good error correcting code, such as the one from Theorem 4.1 (this part depends only on the input). Then it decides on a set of indices in the codeword (this part depends only on the description of the hash function and not on the codeword itself). Lastly it accesses the codeword in exactly those indices picked in the second phase. The output of the hashing process is the set of values assigned by the code to the set of selected indices.[11]

Intuitively, this process achieves 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. The expander random walk will be implemented using the expander from the following theorem.

Theorem 4.2 (implicit in [RVW00])**.** *For every constant $\epsilon > 0$ there exist a constant $d \in \mathbb{N}$ and an algorithm $\mathsf{ExpGen} = \mathsf{ExpGen}_\epsilon$ such that $\mathsf{ExpGen}(N)$ outputs an N-node d-regular graph whose second eigenvalue is smaller than ϵd. Furthermore, ExpGen runs in time $O(N)$.*

[11] The encoding process is similar to previously used methods, e.g. in [NN93].

The following theorem summarizes the properties of the set of indices produced by our hash function. It follows from standard randomness efficiency arguments (see, e.g., survey in [Gol97]).

Theorem 4.3. *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 4.2) 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 4.1), then $\Pr_{\mathsf{rand}}\left[x[I] = y[I]\right] \leq \epsilon$ (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 4.4. *There exists a constant $q > 1$ (the same as in Theorem 4.3) 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 $x \neq y$*

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

4.1 Our Simulator

We next present our simulator whose oracle-RAM complexity is $O(N \log N)$. Our simulator is randomized and succeeds with probability $2^{-\mu N}$ where μ can be any constant.

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 2.1.

Our simulator $\mathsf{QLinSim}$ is described in Figures 4, 5 and 6. Figure 4 is the main procedure; Figure 5 describes the subroutine for hashing and Figure 6 describes "threads". A thread is essentially an additional process (or machine) that reads and writes to shared memory, but also has its own private memory. Since our simulator is sequential, whenever it wants a certain thread to run, it will specify the number of operations to be performed by the thread. Since the thread has its own internal state, in the next time it is called it will proceed from the point where it stopped in the end of the previous call.[12]

As in our previous simulator $\mathsf{PolySim}$, the simulator here is also parameterized by a constant c, which governs the trade-off between communication complexity and error resilience. A larger c will linearly increase the communication complexity, but will (inverse linearly) increase the convergence of the simulator to its best tolerable error rate. 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 2.1), the error correcting code $\mathcal{C}$ (Theorem 4.1) and the hash functions $\mathsf{HashMap}, \mathsf{LinHash}$ (Theorem 4.3 and Corollary 4.4).

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, $\mathsf{QLinSim}$ is similar to our first simulator, presented in Section 3, with a few important differences:

[12] Note that since the input is read from shared memory, it is possible that it changes between calls to the thread, and the thread will need to cope with that.

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 (where ExpGen is from Theorem 4.2, and the parameter r is from Theorem 4.1).
   (b) Run the preprocessing procedure $\mathsf{PreProc}_{\mathsf{ExpSim}}$ for the simulator ExpSim, with $n = \ell_{\mathsf{chunk}} = \gamma \log N$ (where ExpSim, $\mathsf{PreProc}_{\mathsf{ExpSim}}$ and γ are from Theorem 2.1).
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 6).
   (b) Sample $\mathsf{rand}_{1,x}, \mathsf{rand}_{2,x} \leftarrow \{0,1\}^{q \log(cN/r\epsilon)}$.
   (c) Define $\sigma_x := \mathsf{TwoLevelHash}(\mathsf{rand}_{1,x}, \mathsf{rand}_{2,x})$, where $\mathsf{TwoLevelHash}$ is defined in Figure 5.
   (d) Encode $(i, \mathsf{rand}_{1,x}, \mathsf{rand}_{2,x}, \sigma_x)$ using $\mathcal{C}$ to obtain $\omega_x := \mathsf{FastEnc}(i, \mathsf{rand}_{1,x}, \mathsf{rand}_{2,x}, \sigma_x)$, where $\mathcal{C}$ and $\mathsf{FastEnc}$ are from Theorem 4.1 (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 4: An $N \log N$ time simulator.

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 our first simulator, since we separate the input encoding phase from the codeword sampling phase. In addition, we apply a two level hash

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 $I_t := \mathsf{HashMap}_{(cN/r),\epsilon}(2^t \cdot \ell_{\mathsf{chunk}}/r, \mathsf{rand}_1)$, where $\mathsf{HashMap}$ is from Theorem 4.3.

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, where $\mathsf{LinHash}$ is from Corollary 4.4. (rand_2 may be longer than required, in which case only use an appropriate prefix thereof.)

4. Return σ_x.

Figure 5: Subroutine for hashing.

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 4.5 below.

4.2 Analysis

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

Theorem 4.5. *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). We state robustness of this simulator against an unspecified constant error rate. This is in contrast to our first simulator, which we were able to analyze for a specific constant (1/32).

The reason for this discrepancy 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

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

include the headers. We therefore use a separate error correcting code for the headers. Recall that our first simulator combined the headers and the payload into one sequence, and used ExpSim to exchange this entire sequence, which leads to an improved 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 our first simulator. (The preprocessing is protocol independent and can be performed once and for all.) In such case, we can match the error rate of 1/32.

Deterministic Non-Uniform Solution. As is the case with PolySim from Section 3, the simulator QLinSim can be made deterministic in a non-uniform model of computation. The proof is identical to the proof for PolySim in Section 3.3 and we do not repeat it here.

Theorem 4.5 is proven by combining Lemmas 4.8, 4.9 and 4.11 below.

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

Lemma 4.6. *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 the segment $[k \cdot 2^t \cdot \ell_{\mathsf{chunk}} + 1, (k+1) \cdot 2^t \cdot \ell_{\mathsf{chunk}}]$ 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 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 4.7. *The greedy algorithm in* 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 (i.e. not an endpoint) 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 that indeed at most two segments from each division are used.

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

Proof. Preprocessing takes $O(N)$ time by Theorems 4.2, 2.1. 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 4.4), 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 2.1).
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

$$O(N) + O(N/\log N) \cdot O(\log^2 N) = O(N \log N) \ . \qquad \square$$

We proceed by analyzing the communication complexity of our simulator.

Lemma 4.9. *For any $\epsilon = 1/\mathrm{poly}(N)$, the communication complexity of* 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 $O(\log N)$. □

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

Lemma 4.10. *Let $T_1 \neq T_2$ be two transcripts of the same length, and let $\mathsf{TwoLevelHash}^{T_i}(\cdot)$ denote the execution of* 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 4.3 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 4.4 which implies that

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

Applying the union bound, the result follows. □

Finally, we can prove the correctness of QLinSim.

Lemma 4.11. *The simulator* QLinSim *is robust against 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 $2^{-\mu N}$ for any constant μ.

Proof. We follow the steps of the proof of Lemma 3.5 (Section 3). We define the following variables, which are, up to a name change, identical to those in Lemma 3.5:

- 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 the proof of Lemma 3.5 for $\mathsf{PolySim}$ we will use the independence of the hash seeds to bound the probability that above happens "too often". In particular, the bound from Eq. (2) applies in the exact same way, replacing "$\log(N)$" with our chunk length ℓ_{chunk}. It follows that the probability of a false equality happening in more than N/ℓ_{chunk} rounds 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. □

5 Linear-Time Simulator with Polynomial Failure Decay

In this section we present a different approach to interactive coding that allows us 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 an overview of our methods, see Section 1.2.

As in our previous algorithm (see Section 4), 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 2.1.

Our simulator LinSim is defined in Figure 7. Similarly our previous simulators, LinSim is parameterized by a constant c whose value governs the trade-off between communication complexity and tolerable error rate, 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 2.1), the error correcting code $\mathcal{C}$ (Theorem 4.1) and the hash function $\mathsf{LinHash}$ (Corollary 4.4).

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.

5.1 Analysis

The following theorem summarizes the properties of LinSim.

Theorem 5.1. *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 our second simulator (Theorem 4.5), we only state our result for an unspecified error bound. However we can match the error rate of our first simulator (1/32) if polynomial preprocessing is allowed. See the discussion following Theorem 4.5 for details.

Theorem 5.1 is proven by combining Lemmas 5.2, 5.3 and 5.8 below.

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

Lemma 5.2. *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. 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 5.3. *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 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 ExpSim, with $n = \ell_{\mathsf{chunk}} = \gamma \cdot \log N$ (where ExpSim, $\mathsf{PreProc}_{\mathsf{ExpSim}}$ and γ are from Theorem 2.1).
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/ℓ_{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 ω_x over the channel and receive ω_y.
   (e) Decode ω_y into $\mathsf{FastDec}(\omega_y) = (HH'_y, \mathsf{rand}'_y, j)$.
   (f) Compute $\sigma_y := \mathsf{LinHash}_\epsilon((HH_y, HH_x, L, \mathsf{rand}_y, \mathsf{rand}_x, j), \mathsf{rand}'_y)$. (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 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. 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 7: 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 5.4. *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 5.5. *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 4.4)

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

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 5.6 (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*

$$(HH_x^A, HH_y^A, L^A, \mathsf{rand}_x^A, \mathsf{rand}_y^A, i^A) = (HH_y^B, HH_x^B, L^B, \mathsf{rand}_y^B, \mathsf{rand}_x^B, i^B) \ , \tag{3}$$

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

If Eq. (3) *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 5.7. *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 5.8. *The simulator* LinSim *is robust against to a constant fraction of adversarial errors, with failure probability at most* $O(\epsilon N)$*.*

Given Lemma 5.7, the proof of this lemma is follows the same lines as Lemma 4.11 and Lemma 3.5 (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 5.5 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 5.7. 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

[AGHP92] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple construction of almost k-wise independent random variables. *Random Struct. Algorithms*, 3(3):289–304, 1992.

[AGS13] Shweta Agrawal, Ran Gelles, and Amit Sahai. Adaptive protocols for interactive communication. *CoRR*, abs/1312.4182, 2013.

[AS92] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 1992.

[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.

[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.

[CPT13] Kai-Min Chung, Rafael Pass, and Sidharth Telang. Knowledge-preserving interactive coding. In *FOCS*, pages 449–458. IEEE Computer Society, 2013.

[GH13] Mohsen Ghaffari and Bernhard Haeupler. Optimal error rates for interactive coding ii: Efficiency and list decoding. *CoRR*, abs/1312.1763, 2013.

[GHS13] Mohsen Ghaffari, Bernhard Haeupler, and Madhu Sudan. Optimal error rates for interactive coding i: Adaptivity and other settings. *CoRR*, abs/1312.1764, 2013.

[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].

[GN93] Peter Gemmell and Moni Naor. Codes for interactive authentication. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 1993.

[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.

[GSW14] Ran Gelles, Amit Sahai, and Akshay Wadia. Private interactive communication across an adversarial channel. In Moni Naor, editor, *ITCS*, pages 135–144. ACM, 2014.

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

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

[KR13] Gillat Kol and Ran Raz. Interactive channel capacity. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *STOC*, pages 715–724. ACM, 2013.

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

[NN93] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993.

[NSS08] Moni Naor, Gil Segev, and Adam Smith. Tight bounds for unconditional authentication protocols in the manual channel and shared key models. *IEEE Transactions on Information Theory*, 54(6):2408–2425, 2008.

[PCTS00] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Xiaodong Song. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73. IEEE Computer Society, 2000.

[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.