Deterministic and Efficient Interactive Coding from Hard-to-Decode Tree Codes*

Zvika Brakerski†
Weizmann Institute of Science
Rehovot, Israel
zvika.brakerski@weizmann.ac.il

Yael Tauman Kalai
Microsoft and MIT
Boston, MA
yael@microsoft.com

Raghuvansh R. Saxena
Princeton University
Princeton, NJ
rrsaxena@princeton.edu

***Abstract*—The field of Interactive Coding studies how an interactive protocol can be made resilient to channel errors. Even though this field has received abundant attention since Schulman's seminal paper (FOCS 92), constructing interactive coding schemes that are both deterministic and efficient, and at the same time resilient to adversarial errors (with constant information and error rates), remains an elusive open problem.**

An appealing approach towards resolving this problem is to construct an efficiently encodable and decodable combinatorial object called a tree code (Schulman, STOC 93). After a lot of effort in this direction, the current state of the art has deterministic constructions of tree codes that are efficiently encodable but require an alphabet of size logarithmic (instead of constant) in the depth of the tree code (Cohen, Haeupler, and Schulman, STOC 18). We emphasize that we still lack (even heuristic) candidate constructions that are efficiently decodable.

In this work, we show that tree codes that are efficiently encodable, *but not efficiently decodable*, also imply deterministic and efficient interactive coding schemes that are resilient to adversarial errors. Our result immediately implies a deterministic and efficient interactive coding scheme with a logarithmic alphabet (*i.e.*, $1/\log\log$ rate). We show this result using a novel implementation of hashing through *deterministic* tree codes that is powerful enough to yield interactive coding schemes.

***Keywords*-Interactive Coding; Tree Codes; Communication Complexity;**

I. Introduction

In a sequence of groundbreaking works, Schulman [1], [2], [3], defined interactive error-correcting codes. An interactive error correcting code starts with a two party communication protocol that is built to work over the *noiseless* channel, and compiles it to a *noise resilient* two-party protocol that is only a constant factor longer than the noiseless protocol. A protocol is said to be noise-resilient if it works even when a constant fraction of the symbols communicated during the protocol are adversarially corrupted.

Ever since Schulman's works, the hunt for deterministic and efficient interactive codes has remained an elusive open problem. An interactive code is said to be efficient if the 'next message' functions of the noise resilient protocol are efficiently computable given oracle access to the next message functions of the noiseless protocol.

The main line of attack towards getting deterministic and efficient interactive coding schemes is to construct efficiently encodable and decodable combinatorial objects known as *tree codes* [2]. Essentially, a tree code is an error correcting code with an "online" encoding function. That is, for all i, the i^{th} symbol computed by the tree code depends only on the first i symbols of the message it is encoding. The success in this line of attack has been partial. On the one hand, there has been some progress in the direction of getting tree codes that are efficiently encodable [4], [5], [6] but on the other hand, the problem of constructing efficiently decodable tree codes seems to be extremely hard.

A. Our Result

We show that efficient interactive codes can be constructed from tree codes that are not efficiently decodable (but are efficiently encodable). An informal statement of our main result is below (for a formal statement, we refer the reader to the full version [**?**]).

Theorem I.1 (Informal)**.** *There is an efficient and deterministic transformation from an efficiently encodable tree-code to an interactive-coding scheme for two party communication protocols.*

Our result makes a conceptual contribution, showing that efficiently decoding tree codes is not a bottleneck in the path towards getting deterministic and efficient interactive coding schemes. Furthermore, by combining it with known results on tree codes, we obtain interactive coding schemes in regimes where none were known prior to our work. For example, combining our result with the slightly sub-constant rate tree codes of [6] yields a new deterministic and efficient interactive coding scheme with slightly super-linear communication overhead.

Corollary I.2 (Combining our result with [6])**.** *There exists a deterministic interactive coding scheme against adversarial noise that takes a two party protocol of length n and obtains a two party protocol of length $\mathcal{O}(n \log\log n)$ that is resilient to a constant fraction of errors.*

Additionally, although the parameters of the tree codes

*Full version available at https://eccc.weizmann.ac.il/report/2020/137/

†Supported by the Binational Science Foundation (Grant No. 2016726), and by the European Union Horizon 2020 Research and Innovation Program via ERC Project REACT (Grant 756482) and via Project PROMETHEUS (Grant 780701).

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00049

that we use are slightly different, we believe that our techniques can also be combined with those in the work of [5] where the authors show efficiently encodable tree codes based on a conjecture about exponential sums. This will lead to the first full fledged deterministic and efficient interactive coding scheme that has a constant communication blowup and is resilient to a constant fraction of errors based on the same conjecture.

B. Related Work

A lot of work has been done in the field of interactive coding in the past few decades, and our description of the field in the introduction barely scratched the surface. The long line of work [7], [8], [9], [10], [11], [12], [13], [14] (to cite a few) focuses on building better and better (in various aspects) interactive codes in different regimes. We only elaborate on a few that are the closest to ours and refer the reader to [15] for an extensive survey.

Efficient randomized interactive coding schemes: The relaxed problem of constructing efficient *randomized* interactive coding schemes has also received a lot of attention. For our setting of adversarial noise, [9], followed by [16], construct a hash-function based interactive coding scheme. Their coding scheme can also be implemented deterministically in the non-uniform setting, where a non-deterministic "advice" string, linear in the length of the protocol, is given to the communicating parties and the adversary [11] (see also [16]). For the easier setting of random noise, such schemes have been known since [1].

Other interactive coding variants: Many other variants of interactive coding have been studied in the literature, including multi-party interactive coding [17], [18], [19], [20], [21] and list decodable interactive coding [10], [16]. Non-adversarial error models were also considered, most notably the binary symmetric channel (BSC) model, where each bit going over the channel is flipped with some small constant probability [1], [7], [22]. Other models include the insertion deletion model (see *e.g.* [14]).

Tree Codes: Efficiently encodable tree codes are known to be constructible probabilistically with high probability [2], [4], or heuristically based on conjectures on exponential sums [5]. Very recently, an explicit construction with constant error rate and slightly sub-constant $(1/\log\log(n))$ information rate was presented by Cohen, Haeupler and Schulman [6]. Narayanan and Weidner [23] later showed that the tree code of [6] can be efficiently decoded using a randomized algorithm, but only against sub-constant error rate. Also related is the construction of tree codes in sub-exponential time by [24] and the construction of a weaker object called 'potent tree codes' in [7].

C. Our Techniques

It is well known [9] that efficient interactive coding schemes are possible given access to hash functions. In our scheme, we use the same blueprint as [9] but we design a novel way to substitute the *randomized* hash functions with *deterministic* tree codes. It turns out that ensuring that the parties never need to decode the tree code requires the parties to maintain not one, but *several* tree codes in a stack and encode different parts of their state using different tree codes. Matters are further complicated by the fact that the stack needs to be very small almost all the time so that the communication complexity of our simulation does not blow up too much.

We then plug this deterministic tree code based hashing mechanism into the consistency checks of [9] to get an interactive coding scheme. Namely, in our scheme, instead of sending hashes of their local state like in [9], the parties encode their state using various tree codes and exchange these encodings with each other. At first sight, this may seem natural since tree codes, similarly to hash functions, are guaranteed to 'disagree' in many coordinates once the encoded states start becoming different. However, there is a major difference.

Unlike hash functions that are "memoryless" (any invocation of a hash function will detect an inconsistency with high probability), tree codes are static, deterministic objects. It is true that they guarantee a large distance on average, but there may be large 'unprotected' regions of the tree code where they provide no distance guarantees. A lot of work goes behind ensuring that the unprotected regions of different tree codes in our stack do not overlap and we have some protection at all points in our simulation. We next present a detailed overview of our solution. We believe that our approach may be useful in derandomizing other similar interactive tasks.

II. Overview of Our Scheme and Proof

We provide a high level overview of our construction and describe the main ideas behind the proof of security. At a very high level, our construction follows the blueprint of [9], while instantiating the consistency checks using efficiently encodable tree codes instead of hash functions.

A. The Blueprint

In interactive coding, there are two parties, Alice and Bob, that wish to reconstruct the transcript π of a *noiseless* protocol while communicating over a *noisy* channel. This reconstruction is done gradually, where at each point, the parties maintain a local view of π that has been reconstructed thus far. The goal of the parties is to make sure that these (potentially inconsistent) local views eventually converge to the actual transcript π.

In [9], the authors present the following two-layer blueprint for efficient interactive coding schemes:

- **Inner Layer:** The first step in [9] is to break the transcript π into logarithmic sized chunks. In the 'inner layer' of the blueprint, the parties simulate each chunk

of π separately using an inefficient and deterministic (tree-code based) interactive coding scheme. As each chunk is only logarithmic in length, we can afford to simulate these chunks inefficiently while making sure that the overall simulation is efficient.

- **Outer Layer:** In the outer layer, the parties stitch together chunks obtained from the inner layer to get the actual transcript π. Namely, the parties maintain a local transcript, which consists of the sequence of chunks that have been simulated so far, and in each iteration of the outer layer, they execute a *consistency check* to make sure that their local transcripts are consistent. If an inconsistency is noticed then they attempt to fix it.
  The consistency check plays two roles. Firstly, the parties check whether they are 'synchronized' by exchanging with each other the number of chunks that they have simulated so far. If the parties are not in sync, *i.e.*, one party has a simulated more chunks than the other, then the party that has simulated more chunks rewinds one chunk, and the other party keeps their transcript as is.
  Secondly, if the parties are synchronized, they check that their local transcripts (i.e., the sequence of simulated chunks) are equal. To this end, the parties hash their local transcript and exchange these hashes with each other. These hash values come into play when the parties are synchronized, and are a means to detect if their local transcripts are the same. If they are, then the parties proceed to simulate the next chunk. Otherwise, *both* the parties rewind one chunk.
  Analysis: Due to the inner layer, we can assume in our analysis that each iteration (which consists of one chunk simulation followed by a consistency check) is either fully corrupted (and thus contains adversarial content), or not corrupted at all. We need, therefore, to ensure that if at most a constant fraction of iterations are corrupted then the parties output the correct transcript π.
  The analysis is usually done using a potential function, whose goal is to quantify the intuition that "progress" is made in each iteration of the interactive coding scheme. Here, "progress" could either mean that the local views of the parties are now closer to the desired output π, or that some damage done by the adversary in previous iterations is being undone, or (importantly) that the adversary introduced new errors. The latter is considered as "progress" since the budget of the adversary is limited, and the coding scheme benefits from the adversary using up its budget. In order to get an interactive coding scheme with constant blowup in communication, the prime principle to keep in mind is that all communication by the parties needs to be "paid for" by the progress being made. Additionally, if one is interested in constant error rate, then, on average, each error inserted by the adversary is "detected and fixed" after a constant amount of communication.

In [9], these two layers are composed to get the final interactive coding scheme. We emphasize that randomness is only used in the Outer Layer, and in particular in the hashes inside the consistency checks at the end of each iteration.

B. Our Approach: Tree Code Based Consistency Checks

In this work, we replace the randomized consistency checks of [9] with a deterministic tree-code based one. This task is far from trivial. In particular, as we explain below, the number of bits sent in each of our deterministic consistency checks will not be fixed. Rather, in our deterministic solution, the length of each consistency check can be arbitrarily long, depending on the adversarial errors introduced so far. To simplify the description of our consistency checks, we first assume that the parties are always synchronized. In Subsection II-C we show how to deal with synchronization issues.

Basic Idea: A Stack of Tree Codes: Replacing hash functions with tree codes is a natural approach. The purpose of the hashing mechanism is to provide a signal for the event that the two local sequences of chunks disagree. The property of this signal is that once a disagreement occurs, a constant fraction of the iterations result in the parties receiving an indication of this disagreement (with overwhelming probability). Observe that a tree code naturally provides such a mechanism, simply encoding the local view with a tree code does exactly this, since we will still get an indication of the disagreement in a constant fraction of the iterations, and in order to avoid the detection of this discrepancy, the adversary needs to keep investing new errors.

A closer look, however, reveals a very significant difference between detection on a *large but fixed subset* of the iterations, as offered by tree codes, and detection at *every iteration* with high probability, as offered by hash functions. Let us illustrate this using an example.

Suppose that, in some iteration i, the local sequences of chunks of the parties disagree, and even though iteration i was not corrupted, the parties did not detect this disagreement. Such failure can happen in the hash setting due to its probabilistic nature, and in the tree code setting because it only guarantees detection on a large subset of (but not all) iterations. Further, assume (optimistically) that in iteration $i+1$, the disagreement is detected and the parties rewind their local state, reaching the same state as they had in iteration i. Now, in the hash-based solution, we get a second chance to detect the past disagreement, since we use a fresh hash function. The naive tree-code based solution, however, is doomed to repeat the mis-detection due to the deterministic nature of the tree code. This allows the adversary to invest a sufficient amount of errors to place the parties in a vicious loop, and then the adversary can just sit back and watch the protocol never converging.

Our Approach: The parties start the protocol using a "master tree code", and use it to check consistency. Once an inconsistency is detected, rather than simply rewinding one step and then continuing forward with the master alone (which may lead to an infinite loop), our protocol guarantees that the rewinding process removes an actual disagreement. At a high level, this is done by ensuring that the parties keep rewinding until they find a place where their sequences disagree. Such a rewinding procedure is implemented using "backwards tree codes". Namely, as the parties rewind, they send the sequence of chunks, starting from the last, that they are rewinding, to each other, after encoding this sequence using a tree code. The parties will end the rewind stage and continue going 'forward' only after a point of disagreement is found while going backwards.

Applying this approach requires adjustments in order to avoid the following potential problems.

- **Fake disagreement / Fake resolution.** We notice that the disagreement found by the parties may be "fake" in the sense that the parties think there is a disagreement due to adversarial error, even when there is no disagreement. In this case, the parties will continue to rewind all the way to the first chunk, since they will never find a point of disagreement, as none exists.
  We get around this problem by having the parties verify the "reason" for entering the rewind process at each step of the rewind process. Namely, when the parties are rewinding, then, in the consistency check, they not only exchange the encodings of the backwards tree code, but they also exchange the point and value of the tree code that caused the rewinding. We refer to this point and value as the "reason" for rewinding, and denote it by α. If at some point these values agree and there is no reason to go backwards, the parties reverse the backtracking process, one step at a time.
  Adding the reasons to the rewind process ensures that the adversary needs to corrupt the reasons at each step of the rewind process in case of a fake disagreement, thereby getting around the above problem.
  The mirror image of this problem is a situation where the adversary falsely convinces the parties that a disagreement has been found, causing them to terminate the backtracking process prematurely (and thus dooming them to repeat it later on). To handle this, we also consider the reason for terminating the backtracking process (i.e. the inconsistency that has been removed). We denote this reason by β. The parties keep sending the reason, β, for terminating the backtracking process for a number of rounds that is proportional to the number of chunks that have been rewound, thus forcing the adversary to pay in the case of a fake resolution for having to repeat the process.
- **Too many disagreements.** Another case where the adversary's budget fails to pay for the rewinding is in case the adversary inserted a lot of disagreements and it takes too long to correct them one at a time[1]. To understand this case, suppose that the adversary generated a burst of "ancient" errors resulting in a large unprotected region (*i.e.*, a region of the tree code lacking good distance), and then inserted another burst of errors inside this unprotected region. As the region is unprotected, the parties only detect the second burst of errors after a lot of iterations, when they go back and remove the last disagreement in this burst.
  Once they remove this disagreement, the parties resume going forward. We claim that even if the adversary does not insert any more errors, it will take the parties a lot of iterations to get rid of the second burst of errors. This is because they will remove these errors one at a time, and may potentially go forward a lot after any such removal, since the master tree code does not protect them going forward.
  To ensure that error correction happens quickly enough, we "double" the rewind process (for example, if we detect a disagreement $x = 5$ chunks in the past, we will rewind to remove $2x = 10$ chunks from the local transcript). On the one hand, this is only a constant factor, so if there were no ancient errors, then we did not lose too much from this doubling. On the other hand, if there is an ancient burst of errors, then due to the doubling of the rewind process, we make progress fast enough towards correcting all the disagreements in the local view of the parties.
  In our protocol, the doubling process is implemented jointly with the "reasons" mechanism that has been described above. The reason for backtracking (denoted by α) is maintained throughout the $2x$ steps of the backtracking process, whereas the reason for termination (denoted by β) is maintained between steps x and $2x$ (so for a number of steps proportional to the entire backtracking). This way, once we rewound the $2x$ chunks, we are confident that the rewinding process was either uninterrupted, or that the adversary must have invested a sufficient amount of error budget to pay for the excessive steps that were made.
- **Unprotected regions and the stack of tree codes.** Suppose that the adversary inserts many errors at the beginning to create a large "unprotected region". When they are in an unprotected region, it may take many iterations for the parties to detect that a disagreement exists in their local transcript. Consequently, when they eventually do detect this disagreement, the parties may have to rewind many chunks in order to reach this point

[1]Note that disagreements can only be corrected one at a time as after reaching the first point of disagreement, the parties are not even sure whether a second point of disagreement exists or not.

of disagreement. Once this point of disagreement is reached, the parties start going forward again.

However, even when going forward a second time, the parties are still in the unprotected region, and it may take them many iterations to detect any errors that the adversary inserts now. This slow detection means that the adversary wastes a lot of communication for each error he inserts, derailing our interactive coding scheme.

We fix this problem by "patching" the unprotected region of the tree code, as follows. After a disagreement is removed, we add to the original (forward) tree-code, which may be in an unprotected region, a new forward tree-code. We use this new tree-code to encode the suffix of the transcript starting from the point where backtracking ended. This encoding is in addition to the original tree code encoding. As the new tree code is in a protected region (the first levels of any tree code are protected), the parties will quickly detect any new errors inserted by the adversary and continue the simulation.

Observe that adding new tree codes each time a point of disagreement is found can result in the parties having a stack of multiple tree codes. This is because sometimes we will need to "patch the patch" by adding forward tree codes again and again. We refer to this as "the stack of tree codes". As the communication in each iteration will be proportional to the number of tree codes in the stack, we need to make sure to control the size of the stack so as to keep it from blowing up the information rate. That is, we need to decide at some point that a patching tree code served its purpose and can be dropped from the stack. Since the parties do not know where the unprotected regions are, we need some other criterion for dropping the patching tree codes. One natural candidate is to drop the patching tree code at the point where the rewind process started, *i.e.*, if the parties rewound x chunks before going forward again, then the patching tree code will be dropped after x iterations. This idea, however, does not work, and it is possible to create a pathological example where the adversary can make the parties waste a lot of communication in the patching and rewind processes by carefully inserting a small number of corruptions.

The solution is twofold:

- Firstly, we double the length of the patching tree codes. Namely, instead of dropping them after x iterations, the parties drop them after $2x$ iterations. On the one hand, this adds only a constant factor overhead, and is therefore affordable. Moreover, doubling the patching tree codes this way helps us avoid an attack by the adversary where he carefully places corruptions in order to make sure the parties waste a lot of time inside an unprotected region.

  Specifically, suppose that the adversary created an unprotected region of length k on the master tree code using a total of $\Theta(k)$ corruptions. If the adversary does not insert any more corruptions, then the parties will escape the unprotected region in k steps and all will be good. However, if the parties are not doubling the patching tree codes, it is possible for the adversary to carefully insert $\Theta(k)$ more corruptions inside the unprotected region, so that the amount of communication the parties spend before getting out of the unprotected region is $\Omega(k \log k)$, and therefore not affordable. See Appendix A for additional details.
- Doubling the length of the patching tree codes is however not sufficient and in addition, we will also need the patching tree codes deeper in the stack to be more secure than the tree codes higher up in our stack. We do this by adding more and more redundancy to the tree code symbols as we go deeper and deeper in the stack. Specifically, we ensure that the size of the encoding of a tree code grows exponentially with the position of the tree code in the stack.

  This exponential growth in the redundancy is necessary because of the following reasoning. Let $d > 0$ be a constant and consider the case where the stack of tree codes has depth d and the parties are going forward and adding a symbol, say at the i-th chunk of the transcript. As the backwards and forwards tree code alternate in the stack, the parties must have gone back and forth over chunk i of the transcript at least d times, the first time when the stack had depth 1, then again when the stack had depth 2, and so on. When the stack had depth $d' \in [d]$, the parties sent d' encodings while going over this location, one for each tree code in the stack. Thus, the total communication spent on location i is proportional to $1 + 2 + \cdots + d = \Theta(d^2)$.

  This in fact can be the case for a large fraction of the locations as follows. The adversary creates a burst of errors on the master tree-code in order to enter an unprotected region. Then, since it is unprotected, it causes backtracking and the creation of a patching tree code. On the patching tree-code it again creates a burst that creates an unprotected region in the patching code and so forth up to depth d. The crucial point is that when we pop the patching tree code, the transcripts of the parties are completely corrupted (initially by the master burst, then the second level burst and so on), so that the protocol might never leave the unprotected region. Therefore it is possible to restart the process without investing in a new burst. Tuning the lengths of the bursts respective to the properties of the tree code allows to make the cumulative length of all bursts at level

d' to be the same across all d'. Note that the cost of each error in a depth-d' burst is d'. However, the communication incurred by such a location is roughly d'^2 as explained above. The gap between the communication and number of errors thus grows with d, and since this can happen for any constant d, we cannot guarantee any constant fraction of corruptions.

Adding redundancy to tree codes down the stack prevents such irregularities, since now both the communication and the computation will be dominated by the exponentially growing term.

In conclusion, our consistency check maintains a stack of tree codes consisting of the master code, backtracking codes, and patching codes. We use the term forward tree codes to refer to the master and the patching tree codes together. The number of bits sent in each consistency check depends on the number of tree codes in the stack. Importantly, the consistency check only requires *efficiently encodable* tree codes and does not require any randomness.

C. Synchronization

Recall that we assumed the parties are always synchronized in Subsection II-B when describing our tree code based consistency checks. We next remove this assumption and show how the parties can synchronize so as to complete the description of the Outer Layer of our protocol.

Recall that synchronization is needed in [9] because if the parties are not synchronized, then they are simulating different chunks of the noiseless transcript and the resulting communication is meaningless. Moreover, in this case, the backtracking process described above is not effective, as there, the parties backtrack together, which will keep them unsynchronized. Therefore, we need to add to our consistency checks a way to check that the parties are synchronized.

The presence of a stack of tree codes instead of a single hash function and the fact that the communication complexity of an iteration is not fixed make our synchronization procedure more complicated than that in [9]. Firstly, in order to detect which party is ahead of the other, the parties can no longer rely on the length of their local sequence of chunks. Indeed, this length is not a good estimate because the parties can go both forward and backwards in the sequence. Instead, the parties determine who is further ahead based on the *total* number of chunks, both forward and backward, simulated so far and this number is exchanged as a part of the synchronization information. In addition, they exchange the number of tree codes they have in their stack, and the number of chunks in each party's local sequence at the time when each tree code was added to the stack. This synchronization information adds only a constant multiplicative overhead to the number of bits sent in the consistency checks, and therefore blows up the length of our interactive coding scheme by only a constant factor.

Furthermore, because the communication in each iteration is different, when the parties detect that they are out of sync (say Alice is ahead of Bob), then instead of Alice going back one iteration in time, as in [9], she needs to go back an amount proportional to the communication in this iteration. For example, suppose that Alice and Bob detect a synchronization error in iteration i, when the length of the synchronization information received by Alice from Bob is b bits. Also suppose that in all the iterations 1 through $i-1$ Alice sent a bits of synchronization information to Bob. Then, in iteration i, Alice will need to go back (roughly) b/a iterations, so that the total amount of communication that she rewound is around $a \cdot \lceil b/a \rceil = b$. Rewinding in this way ensures that the amount of communication it takes to get back in sync is upper bounded by a constant times the amount of errors invested by the adversary to make the parties fall out of sync.

Finally, just like in [9], if the parties are synchronized, then they continue the consistency check as described in the foregoing section.

D. The Model LONG

Observe that the number of bits required to encode our synchronization information, and the number of tree code encodings exchanged by the parties, depend on the number of tree codes in the parties' stack. Correspondingly, the number of bits sent by the parties in the consistency check in the Outer Layer will vary from iteration to iteration. In order to capture this neatly, we first present our protocol in an artificial model, called model LONG. We then show how to convert any protocol in model LONG to one in the standard two party model.

In model LONG, the protocol proceeds in "iterations" of $P+1$ rounds, where P is the time it takes to simulate one chunk of the noiseless transcript (i.e. P is logarithmic in the total length of the underlying transcript π). Each iteration consists of P "standard/short" rounds where parties send each other one symbol per round, followed by one "long" round where the parties may send arbitrarily long messages. The long round is used to send the consistency check information in our Outer Layer, where the parties may be sending a lot of bits based on the number of tree codes currently in their stack. In model LONG, we consider adversaries that either corrupt an entire iteration, and are charged by the communication complexity of that iteration, or leave the iteration completely uncorrupted. This is similar to the adversaries in [9], except that for us, the communication complexity of an iteration is not fixed, and depends on the adversarial error.

Transformation to the Standard Model: We convert a protocol in model LONG to a protocol in the standard model, with essentially the same error resilience properties (up to

constant factors). The fact that the long messages exchanged after every P rounds in model LONG are arbitrarily long makes this part quite tedious[2]. We do this in two steps:

1) First, we convert the protocol in model LONG into another protocol in model LONG, which has the guarantee that all the long messages are of the same fixed length which is a constant times P.
2) Then, we convert the protocol obtained in Step 1 above into one in the standard model.

Observe that Step 2 is quite straightforward and is very similar to [9] described in Subsection II-A, and it simply applies a deterministic tree code based interactive coding scheme to each iteration separately. Since each iteration is only of length $\mathcal{O}(P)$ after Step 1, which is logarithmic in the length of the noiseless transcript, the tree codes required in this step are computationally efficient.

However, Step 1 is tedious, and is roughly carried out as follows. When a party needs to send a long message, the party first encodes the message using a standard error correcting code resilient to insertions and deletions. Then the party breaks this encoded message (which may be way too long) into blocks, each of length P. The parties will then send these blocks one at a time, along with some metadata, which includes which block number it is, and whether it is the last block or not, *etc*. This metadata will help the receiving party stitch the individual blocks together in the right order.

The reason we use error correcting codes resilient to insertion and deletion errors is to simulate the guarantee in model LONG that says that corrupting any part of the long message or the P standard rounds before it requires the adversary to spend a number of errors proportional to the length of the long message. Error correcting codes resilient to insertions and deletions provide this abstraction (up to constant factors) even if certain blocks are lost in the transmission and reconstruction.

However, even these error correcting codes do not make sure that corrupting any of the P standard rounds requires as many corruptions as the length of the long message. To ensure this, we repeat these P standard rounds before every block in the long message and only proceed if all these repetitions are consistent with each other, *i.e.* they yield the same transcript. If there are blocks where these P rounds are not consistent with each other, we rewind one block and repeat to (hopefully) get consistency. Ensuring that the repetitions are mutually consistent in turn ensures that the adversary needs to invest a lot of corruptions if he wants to corrupt the P standard rounds. We refer the reader to the full version for details.

[2]Dealing with variable message-length in interactive coding was already considered in previous works [25] (although their focus was different, since their goal was to preserve the round complexity).

III. Our Protocol in Detail

We now describe our protocol for converting a 2-party protocol Π into an error resilient protocol Π' in more detail. We present the protocol in model LONG since we view the conversion to the standard model as tedious and not as insightful.

As motivated in Section II, the protocol Π' breaks the protocol Π into small chunks, and simulates the protocol Π chunk by chunk. Before adding each simulated chunk to the local transcript π, the protocol runs a synchronization mechanism to check that the parties are synchronized, and if so it runs a consistency mechanism to detect and fix errors in the simulation of these chunks. Namely, if the parties detect an error in one of the chunks they simulated, they start going 'backwards' over the chunks to find the source of this error. When the parties are going backwards over the chunks, they add the transcripts of these chunk to a 'backwards transcript', denoted by ψ, that the protocol maintains. This means that at any point in the protocol, the parties will be maintaining two transcripts π and ψ, where the transcript ψ is different from the empty transcript ε only if the parties are going backwards over the chunks to find a source of an error they detected.

Thus, the protocol Π' consists of three components:

1) Chunk, where the parties simply simulate the next P rounds of the underlying protocol Π (recall that P is logarithmic in the total length of the underlying transcript π).
2) Sync, which is the synchronization mechanism that was outlined in Subsection II-C, and which we elaborate on in Subsection III-B.
3) AddSym, which is the consistency mechanism that was outlined in Subsection II-B, and is elaborate on in Subsection III-A. This subroutine decides whether to add the symbol computed by Chunk to the simulated transcript π, or to remove the last chunk of π and add it to the backwards transcript ψ, in the case where the parties are backtracking.

Our consistency mechanism AddSym utilizes a stack of tree codes maintained by the parties.

Structure of the stack of tree codes: The stack of tree codes maintained by the parties has two kinds of tree codes, in alternation: forward tree codes and backward tree codes. The forward tree codes encode suffixes of the transcript π while the backward tree codes encode the transcript ψ. We number the trees in the stack from the bottom, thus odd-indexed stack entries refer to forward tree-codes, while even numbered entries refer to backwards tree codes. The so-called master tree-code therefore has the index 1. This implies that when the number of tree codes in the stack is odd, then the last tree code in the stack is a forward tree code and the protocol is going forward on the transcript π, and vice versa.

We note that each of the tree codes in the stack is actually the same tree code, but is used to encode a different string. Each forward tree code encodes some suffix of the transcript π, while each backwards tree code encodes some part of π in a backwards order (denoted by ψ).

We describe our protocol Π' in Algorithm 1, the consistency mechanism AddSym in Algorithm 2, and the synchronization mechanism Sync in Algorithm 3. In our protocol the parties maintain a list of states $\boldsymbol{\mathcal{S}}$. Each element in the list $\boldsymbol{\mathcal{S}}$ is described by a tuple $(\mathscr{R}, \pi, \psi, p)$, where $\mathscr{R}$ is the stack of tree-codes and each element in this stack is described via four variables (r, t, α, β) (which we elaborate on in Subsection III-A below), π is the forwards simulated transcript, ψ is the backwards simulated transcript (which is $\perp$ if the parties are going forwards), and p is the length of the long message that was sent during that state. Keeping this entire list of states $\boldsymbol{\mathcal{S}}$ is important, since if the parties go out of sync, then they will need to backtrack to an earlier state.

We would like to highlight an important difference in the way forward and backward tree codes are used in Algorithm 1. When going forward, we use *all* forward tree codes in the stack to encode the transcript (we denote the subset of froward tree codes in the stack by $\mathscr{R}_{\mathrm{F}}$). In contrast, a backwards tree code is only in use when it is on the top of the stack.

Also recall from Subsection II-B that the tree codes need to have more and more redundancy as the stack of the parties grows deeper and deeper. This is implemented (in Line 12) by padding the long message sent by the parties to length $O(P) \cdot 2^{|\mathscr{R}|} + p/2$. The first term captures the redundancy while the second term is needed to ensure that the parties can resynchronize if the adversary inserts corruptions and makes them unsynchronized. Its significance will be explained in Subsection III-B.

Notation: We use $|\boldsymbol{\mathcal{S}}|$ to denote the number of elements in the list $\boldsymbol{\mathcal{S}}$. We define $|\mathscr{R}|$ similarly. For $1 \le i \le |\boldsymbol{\mathcal{S}}|$, the notation $\boldsymbol{\mathcal{S}}[i]$ denotes the i^{th} element in $\boldsymbol{\mathcal{S}}$. When $i = |\boldsymbol{\mathcal{S}}|$, we sometimes use $\boldsymbol{\mathcal{S}}.last$ instead of $\boldsymbol{\mathcal{S}}[|\boldsymbol{\mathcal{S}}|]$. The variable π in the i^{th} element of $\boldsymbol{\mathcal{S}}$ is denoted by $\boldsymbol{\mathcal{S}}[i].\pi$. The corresponding quantities for other fields in $\boldsymbol{\mathcal{S}}[i]$ and for the list $\mathscr{R}$ are defined analogously. When we wish to add an element $e = (\mathscr{R}, \pi, \psi, p)$ to $\boldsymbol{\mathcal{S}}$, we denote this using $\boldsymbol{\mathcal{S}}.\textsc{Add}(e)$. The element e is then added at the end of the list. Likewise for $\mathscr{R}$. When we wish to remove the last element from $\boldsymbol{\mathcal{S}}$, we write $\boldsymbol{\mathcal{S}}.\textsc{Rem}()$. After this operation, the list has one less element. We write $\boldsymbol{\mathcal{S}}.\textsc{Rem}(i)$ to denote the operation of removing the last i elements in the list. Likewise for $\mathscr{R}$. The notation $\mathscr{R}_{\mathrm{F}}$ will be used to denote the sublist of $\mathscr{R}$ that contains all the elements in odd positions in $\mathscr{R}$. This notation derives from the fact that the odd positions in $\mathscr{R}$ are occupied by forward tree codes. Also, we use the variables $\mathscr{R}$, π, ψ, p, and ℓ freely throughout the protocols we describe. These are our global variables and can be accessed from anywhere in the protocol. The variable ℓ will be reserved for the length of the message sent by the parties in Line 12.

Algorithm 1 Our interactive coding scheme (Alice's side).

Input: An input $x^A \in X^A$.
Output: An element in Y^A.

1: Initiate a structure $\boldsymbol{\mathcal{S}} \leftarrow [(\mathscr{R}, \pi, \psi, p)]$, where $\mathscr{R} \leftarrow [(0, 0, \perp, \perp)]$ is the initialized stack of tree-codes, π is the forward transcript and ψ is the backwards transcript, initialized to $\pi = \psi = \varepsilon$ and $p = 0$.
2: **for** $i \in [\mathcal{O}(|\Pi|/P)]$ **do**
3: $(\mathscr{R}, \pi, \psi, p) \leftarrow \boldsymbol{\mathcal{S}}.last$.
4: $\sigma \leftarrow \textsc{Chunk}()$.
5: **if** $|\mathscr{R}|$ is odd (*i.e.*, the party is going forward) **then**
6: Γ is the tree code encoding of all the forward tree codes in the stack $\mathscr{R}$; *i.e.*, $\Gamma \leftarrow \left[\mathsf{TC}((\pi \| \sigma)_{>r}) \textbf{ for } (r, \cdot, \cdot, \cdot) \in \mathscr{R}_{\mathrm{F}}\right]$.
7: **else** (the party is going backwards)
8: $\sigma \leftarrow \pi[|\pi|]$. (In this case we do not use the simulated chunk from Line 4, rather we are looking for a disagreement in our forward transcript π.)
9: $\Gamma \leftarrow [\mathsf{TC}(\psi \| \sigma)]$.
10: **end if**
11: $\mathscr{P} \leftarrow (\mathscr{R}, |\pi|, |\psi|, p)$.
12: Send $(\mathscr{P}, |\boldsymbol{\mathcal{S}}|, \Gamma)$ and receive $(\tilde{\mathscr{P}}, |\tilde{\boldsymbol{\mathcal{S}}}|, \tilde{\Gamma})$ as elements of Σ^*, and pad this message by zeros to ensure that it consists of $\ell \leftarrow O(P) \cdot 2^{|\mathscr{R}|} + p/2$ symbols from Σ. Let $\tilde{\ell}$ be the number of symbols received.
13: **if** $\textsc{Sync}()$ **then**
14: $\textsc{AddSym}(\sigma, \Gamma, \tilde{\Gamma})$.
15: $\boldsymbol{\mathcal{S}}.\textsc{Add}(\mathscr{R}, \pi, \psi, \ell)$.
16: **end if**
17: **end for**
18: $(\mathscr{R}, \pi, \psi, p) \leftarrow \boldsymbol{\mathcal{S}}.last$.
19: Output $g^A(x^A, \pi[1 : T/P])$ interpreting $\pi[1 : T/P]$ as an element of Σ^{2T}.

A. The Consistency Mechanism

Our consistency mechanism, AddSym, is described formally in Algorithm 2, and in what follows we provide an informal description.

The algorithm takes as input the new transcript chunk σ, and the consistency information $\Gamma, \tilde{\Gamma}$, where Γ represents the player's "local" consistency information (computed in Line 6 or Line 9 of Algorithm 1), and $\tilde{\Gamma}$ is the other party's consistency information. For our purposes, it only matters whether the two are equal or not in order to determine consistency.

We associate with each tree-code in the stack four parameters (r, t, α, β), where the parameter r corresponds to its root, which denotes the position in π where this tree-code was initiated. The parameter t should be understood the "turning point", which is the point in π where the next

Algorithm 2 The Protocol $\textsc{AddSym}(\sigma, \Gamma, \tilde{\Gamma})$.

20: **if** going forward **then**
21: Add σ to the end of the transcript π.
22: **if** consistency mismatch ($\Gamma \neq \tilde{\Gamma}$) **then**
23: Set the turning point in the last forwards tree code in the stack to be $|\pi|$ ($\mathscr{R}.last.t \leftarrow |\pi|$).
24: Add a new backwards tree code to the stack of tree codes $\mathscr{R}$. This new backward tree-code is associated with parameters (r, t, α, β), where $r = |\pi|$, $t = 0$, $\alpha = (\Gamma, \tilde{\Gamma})$, and $\beta = \perp$.
25: **else** (remove tree-codes that are no longer needed)
26: $d^* \leftarrow$ smallest odd number such that $|\pi| - \mathscr{R}[d+2].r \geq 2(\mathscr{R}[d].t - \mathscr{R}[d+2].r)$. If none exists, set $d^* \leftarrow |\mathscr{R}|$. (This corresponds to removing a patching tree code, together with the backward tree code proceeding it, after it is 'doubled'.)
27: $\mathscr{R}.\textsc{Rem}(|\mathscr{R}| - d^*)$.
28: **end if**
29: **else** (the party is going backwards)
30: Add σ to the end of the backwards transcript ψ.
31: Remove the last chunk from forwards transcript ($\pi \leftarrow \pi_{<|\pi|}$).
32: **if** $|\psi| = 2(\mathscr{R}.last.r - \mathscr{R}.last.t)$ **then**
33: This corresponds to the case when the backwards tree code has doubled and ψ is double its length where the discrepancy was found. We erase the reasons and the backwards transcript, *i.e.* $\mathscr{R}.last.\alpha, \mathscr{R}.last.\beta \leftarrow \perp$ and $\psi \leftarrow \varepsilon$.
34: Add a forward tree code (r, t, α, β) where $r = |\pi|$, $t = 0$ and the reason $\alpha = \beta = \perp$, *i.e.*, $\mathscr{R}.\textsc{Add}(|\pi|, 0, \perp, \perp)$.
35: **else**
36: **if** we backtracked all the way, *i.e.*, $\pi = \varepsilon$ **then**
37: Re-initialize the tree code stack $\mathscr{R} \leftarrow [(0, 0, \perp, \perp)]$ and set $\psi \leftarrow \varepsilon$.
38: **else**
39: We check if we rewound past the roots of any forwards tree codes. If so, we delete these and the backwards tree codes right before them. Let d^* be the largest (possibly 0) such that $\forall d' \in [d^*] : \mathscr{R}[|\mathscr{R}| - 2d' + 1].r = |\pi|$.
40: Delete positions d' from $\mathscr{R}$ for all $|\mathscr{R}| - 2d^* \leq d' < |\mathscr{R}|$.
41: **if** $\Gamma \neq \tilde{\Gamma}$ **and** $\mathscr{R}.last.t = 0$ **then**
42: $\mathscr{R}.last.t \leftarrow |\pi|$.
43: Let β be the reason for the discrepancy. Namely, $\mathscr{R}.last.\beta \leftarrow (\Gamma, \tilde{\Gamma})$. Note that $|\Gamma| = |\tilde{\Gamma}| = 1$ in this case (of going backwards).
44: **end if**
45: **end if**
46: **end if**
47: **end if**

tree code was pushed into the stack. For backwards tree codes, we also maintain the "reasons" α, β as described in Subsection II-B, whereas for forward tree-codes $\alpha = \beta = \perp$.

Operations on the stack of tree codes: We next describe the operations performed on this stack of tree codes.

- The first and the most basic operation that the parties will need to perform on the stack is to add the transcripts of the simulated (or rewound) chunks, to the tree codes in the stack. When the parties are going forward, then this operation is simply adding the transcript, σ, of the chunk just simulated to π (Line 21). On the other hand, when the parties are going backwards, then this operation involves removing the transcript, σ, of the most recent chunk from π and adding it to the backwards transcript (Line 30 and Line 31)
- Besides adding the transcript chunks to the tree codes present in the stack, we will also need to add fresh tree codes to the stack. Recall from Subsection II-B that any new tree codes added to the stack has a reason associated with it. In our protocol, when we start going backwards, we save a reason α for this backtracking, and when removing a discrepancy we save the reason β for wanting to terminate the backtracking. Let us explain this mechanism in more detail.
  The parties will switch from going forward to backtracking as soon as they see that the tree code encoding they computed (Γ) is different from the tree code encoding that they received ($\tilde{\Gamma}$). The length of the transcript at this point as considered as the "turning point" t of the current forward progress and will be saved in the proper variable of the current forward tree code Line 23. The same value is considered as the "root value" r of the new backwards tree code and will be marked as such. The "offending" $(\Gamma, \tilde{\Gamma})$ will be marked as the backtracking reason (Line 24).
  On the other hand, when the parties see a discrepancy in their encodings while going backward, they do not turn and start a forward tree code immediately. As explained in Subsection II-B, the parties instead record this point of discrepancy in the variable t (Line 42), store the discrepancy as their reason (Line 43) in a variable β, and turn after the transcript sent over the backward tree code is double of what it was when the discrepancy was found. If this happens, then Line 33 and Line 34 are executed, where the parties add a new forward tree code (and set ψ to ε as it is no longer needed).
- Finally, we discuss dropping tree codes from the stack of tree codes. There are two points where the parties drop tree-codes from the stack.
  The first point is in Line 27, where the parties pop a forward tree code after 'doubling' it, and popping all the tree codes starting from the forward tree code that was doubled. To check whether there is a forward tree code that has been doubled, the parties go over all odd

$i \in |\mathscr{R}|$ (recall that forward tree codes are stored in odd positions in the stack) and see if $2(\mathscr{R}[i].t - \mathscr{R}[i+2].r) \leq |\pi| - \mathscr{R}[i+2].r$, *i.e.*, they see if the length of the current transcript π is at least double of where the last forward tree turned to a backward, *i.e.*, $\mathscr{R}[i].t$, measured from the current root $\mathscr{R}[i+2].r$.

The other point where parties pop a tree code is when they turn forwards from backward, say at point r, and the protocol reaches a point where $|\pi| \leq r$. When this happens, they pop both of these tree codes from the stack (Line 40). Also, if the parties go back all the way to $|\pi| = 0$, they drop all the tree codes from the stack and reinitialize $\mathscr{R}$ and ψ (Line 37).

We note that the parties do not need a backward tree code in the stack once they add a new forward tree code on top of it. Nonetheless, we keep these backward tree codes (for simplicity) as they only cost us a constant factor overall.

B. The Synchronization Mechanism

We now provide a detailed description of the synchronization mechanism outlined in Subsection II-C.

Algorithm 3 The Synchronization Mechanism SYNC.

48: **if** $|\boldsymbol{\mathcal{S}}| = |\tilde{\boldsymbol{\mathcal{S}}}|$ **and** $\mathscr{P} = \tilde{\mathscr{P}}$ **then**
49: Return `True`.
50: **else if** $|\boldsymbol{\mathcal{S}}| = |\tilde{\boldsymbol{\mathcal{S}}}|$ **or** $10\tilde{\ell} < \ell$ **then**
51: $\boldsymbol{\mathcal{S}}$.REM().
52: **else if** $|\boldsymbol{\mathcal{S}}| > |\tilde{\boldsymbol{\mathcal{S}}}|$ **then**
53: $\boldsymbol{\mathcal{S}}$.REM$(\min(\mu, |\boldsymbol{\mathcal{S}}| - |\tilde{\boldsymbol{\mathcal{S}}}| + \mathbb{1}(10\ell < \tilde{\ell})))$ where μ is the smallest integer that satisfies $\sum_{h=1}^{\mu} \boldsymbol{\mathcal{S}}[|\boldsymbol{\mathcal{S}}| + 1 - h].p > 10(\ell + \tilde{\ell})$.
54: **end if**
55: Return `False`.

Recall that our synchronization mechanism largely mimics the synchronization mechanism of [9] for the most part, but has some non-trivial adaptations.

We recall again that in the synchronization mechanism of [9], the parties maintain a 'state' of the protocol. In [9], the length of this state is just the length of the transcript simulated so far. After simulating a fresh chunk of the protocol, the parties share the length of their state with each other. If these lengths are the same, then both parties have local transcripts of the same length, and they continue the protocol as normal. On the other hand, if one party has a longer transcript than the other party, then, the party that is 'ahead' will rewind one chunk at time, so that both the parties have transcripts of the same length.

The reason the parties share the length of the transcript simulated so far with each other is that this length is a natural measure of the amount of progress the parties have made in the simulation, *i.e.*, it is true that if a party in [9] has a longer transcript than the other party, and this party rewinds one chunk, then the two parties will only get closer to each other. This is no longer the case in our protocol, as the party who has a longer transcript may or may not be ahead of the other party, depending on whether the parties are going forwards or backwards.

The History Stack: In the synchronization mechanism, we will sometimes require the parties to "go back in time" to a prior state in the execution. In order to enable this, each party maintains a stack of the entire history of its complete local state. The history stack is denoted by $\boldsymbol{\mathcal{S}}$ and each entry in $\boldsymbol{\mathcal{S}}$ contains a tuple $(\mathscr{R}, \pi, \psi, p)$, which will allow to resume the execution from the given point in history. The parameter p is used to throttle the communication rate and will be discussed further below. The history stack is updated every time a symbol is added to any of the tree codes (Line 15). The total number of elements in $\boldsymbol{\mathcal{S}}$ therefore serves as our analogue of the length of the simulated transcript of [9].

The Synchronization Stamp $\mathscr{P}$: For the parties to be synchronized, they need to agree on the number of algorithmic steps made so far, which is captured by $|\boldsymbol{\mathcal{S}}|$ as explained above. However, this by itself is insufficient. The fact that our rewind mechanism deals with a stack with multiple tree codes creates an additional complication in the synchronization mechanism. Consider a situation where Alice and Bob have the same transcript π but two different stacks of tree codes. Specifically, assume that the set of the roots of the tree codes are not the same for Alice and Bob. In this case, even though the transcripts are the same for Alice and Bob, the fact that the roots are different means that the tree code encodings are potentially different, and the parties will not be able to continue with the simulation thinking that errors have happened.

In order to get around this problem, we have the parties exchange an additional "synchronization stamp" $\mathscr{P}$: a few parameters of their stack of tree codes that will allow to ensure synchronization. The stamp $\mathscr{P}$ includes $\mathscr{R}$, the stack of tree codes, $|\pi|$ and $|\psi|$, the lengths of the forward and backward transcripts, and the aforementioned variable p. Note that, in particular, $\mathscr{P}$ contains the roots of all the tree codes and the lengths of the transcript and thus, the problem in the foregoing paragraph does not arise.

Synchronization: To summarize, after simulating every chunk the parties send the values $|\boldsymbol{\mathcal{S}}|$ and $\mathscr{P}$ to each other along with the encoding on the tree codes (Line 12). If these values match, then the parties add a symbol to their (either forward or backward) transcript (Line 49).

If the value of $|\boldsymbol{\mathcal{S}}|$ agrees with the value received but the value of $\mathscr{P}$ does not, then the parties deduce that they have added the same number of symbols but there was a discrepancy in one of the tree codes that they added. (Both) the parties rewind one step in this case hoping to revert to a

state where the values of $\mathscr{P}$ match[3]. This is done in Line 51. We next mention a subtlety of our protocol that makes Line 51 work. This subtlety arises because the number of bits sent by the parties in Line 12 is a function of the state the parties are in, and thus may be different for different states. Consider a situation where a state for which the parties communicate a lot is followed immediate by a state for which the parties communicate very little. If the parties wrongly (due to corruptions) decide to execute Line 51 in the iteration with little communication, then they actually end up rewinding a lot of communication due to a small number of errors. This is problematic, and the way we get around this problem is by ensuring that the communication in adjacent states differs by a factor of at most 2 in Line 12.[4]

Lastly, the value of $|\boldsymbol{\mathcal{S}}|$ is greater than the value received by the party, then the party thinks that they are ahead of the other party and would like to rewind. As in the foregoing paragraph, we would like to rewind a number of states roughly 'equivalent' to the communication in this round. As the amount of communication may be different in different iterations, there is no direct correspondence between the amount of communication in a state to the number of states. This is where the variable p comes in. The variable p for a state in $\boldsymbol{\mathcal{S}}$ stores the amount of communication done by the parties to reach that state. In Line 53, we remove a number of states so that the sum of the corresponding p values is bounded by a constant times the amount of communication in this iteration.

IV. Our Approach for Analysis

The analysis of our scheme follows a similar blueprint to prior works, and uses a potential function to measure progress. While prior works which broke the protocol into *a-priori fixed* chunks, and proved that the potential goes up (at a sufficient rate) in each chunk, such a simplistic approach does not work in our case. In our analysis, we break the protocol into chunks whose lengths are not a-priori fixed, and are influenced by the adversary.

Intuitively, the reason is that the adversary can invest a lot of errors and cause the protocol to enter a long unprotected region, and in this region no progress is being made (in the sense that the adversary does not use his error budget, nor are the parties transcripts closer to agreement). Thus, in our analysis we define the chunks based on when new patching tree-codes were added and removed, and then use a potential function approach to argue that progress is being made at a sufficient rate (i.e., the potential function increases). We refer the reader to the full version for details [**?**].

[3]There exists such a state because both the parties start with the same state.

[4]The condition $10\tilde{\ell} < \ell$ before Line 51 arises due to a technicality in our analysis.

References

[1] L. J. Schulman, "Communication on noisy channels: A coding theorem for computation," in *Foundations of Computer Science (FOCS)*, pp. 724–733, IEEE, 1992. 1, 2

[2] L. J. Schulman, "Deterministic coding for interactive communication," in *Symposium on Theory of computing (STOC)*, pp. 747–756, ACM, 1993. 1, 2

[3] L. J. Schulman, "Coding for interactive communication," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 1745–1756, 1996. 1

[4] M. Peczarski, "An improvement of the tree code construction," *Inf. Process. Lett.*, vol. 99, no. 3, pp. 92–95, 2006. 1, 2

[5] C. Moore and L. J. Schulman, "Tree codes and a conjecture on exponential sums," in *Innovations in theoretical computer science (ITCS)*, pp. 145–154, ACM, 2014. 1, 2

[6] G. Cohen, B. Haeupler, and L. J. Schulman, "Explicit binary tree codes with polylogarithmic size alphabet," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018* (I. Diakonikolas, D. Kempe, and M. Henzinger, eds.), pp. 535–544, ACM, 2018. 1, 2

[7] R. Gelles, A. Moitra, and A. Sahai, "Efficient and explicit coding for interactive communication," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pp. 768–777, IEEE, 2011. 2

[8] M. Braverman and A. Rao, "Towards coding for maximum errors in interactive communication," in *Symposium on Theory of computing (STOC)*, pp. 159–166, ACM, 2011. 2

[9] Z. Brakerski and Y. T. Kalai, "Efficient interactive coding against adversarial noise," in *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pp. 160–166, IEEE, 2012. 2, 3, 6, 7, 10

[10] M. Braverman and K. Efremenko, "List and unique coding for interactive communication in the presence of adversarial noise," in *Foundations of Computer Science (FOCS)*, pp. 236–245, 2014. 2

[11] Z. Brakerski, Y. T. Kalai, and M. Naor, "Fast interactive coding against adversarial noise," *Journal of the ACM (JACM)*, vol. 61, no. 6, p. 35, 2014. 2

[12] R. Gelles, A. Moitra, and A. Sahai, "Efficient coding for interactive communication," *IEEE Transactions on Information Theory*, vol. 60, no. 3, pp. 1899–1913, 2014. 2

[13] K. Efremenko, R. Gelles, and B. Haeupler, "Maximal noise in interactive communication over erasure channels and channels with feedback," *IEEE Transactions on Information Theory*, vol. 62, no. 8, pp. 4575–4588, 2016. 2

[14] M. Braverman, R. Gelles, J. Mao, and R. Ostrovsky, "Coding for interactive communication correcting insertions and deletions," *IEEE Transactions on Information Theory*, vol. 63, no. 10, pp. 6256–6270, 2017. 2

[15] R. Gelles, "Coding for interactive communication: A survey," *Foundations and Trends® in Theoretical Computer Science*, vol. 13, no. 1–2, pp. 1–157, 2017. 2

[16] M. Ghaffari and B. Haeupler, "Optimal Error Rates for Interactive Coding II: Efficiency and List Decoding," in *Symposium on Foundations of Computer Science (FOCS)*, FOCS, pp. 394–403, 2014. 2

[17] S. Rajagopalan and L. J. Schulman, "A coding theorem for distributed computation," in *Symposium on the Theory of Computing (STOC)*, pp. 790–799, 1994. 2

[18] A. Jain, Y. T. Kalai, and A. B. Lewko, "Interactive coding for multiparty protocols," in *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, pp. 1–10, ACM, 2015. 2

[19] W. M. Hoza and L. J. Schulman, "The adversarial noise threshold for distributed protocols," in *Symposium on Discrete Algorithms (SODA)*, pp. 240–258, 2016. 2

[20] K. Efremenko, G. Kol, and R. Saxena, "Interactive coding over the noisy broadcast channel," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 507–520, ACM, 2018. 2

[21] R. Gelles, Y. T. Kalai, and G. Ramnarayan, "Efficient multiparty interactive coding for insertions, deletions and substitutions," *CoRR*, vol. abs/1901.09863, 2019. 2

[22] R. Gelles, B. Haeupler, G. Kol, N. Ron-Zewi, and A. Wigderson, "Towards optimal deterministic coding for interactive communication," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1922–1936, Society for Industrial and Applied Mathematics, 2016. 2

[23] A. K. Narayanan and M. Weidner, "On decoding cohen-haeupler-schulman tree codes," *CoRR*, vol. abs/1909.07413, 2019. 2

[24] M. Braverman, "Towards deterministic tree code constructions," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pp. 161–167, ACM, 2012. 2

[25] K. Efremenko, E. Haramaty, and Y. Kalai, "Interactive coding with constant round and communication blowup," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 25, p. 54, 2018. 7

APPENDIX A.
THE NEED FOR DOUBLING PATCHING TREE CODES

We show that when patching tree codes are not doubled (i.e. terminated at the point when the corruption was initially noticed), it is possible for the adversary to make k corruptions that takes $\Omega(k \log k)$ to correct. Our example uses a stack of depth 3, i.e. there is at most one patching tree code in the stack at any point in time.

The starting point of the example is a large unprotected region in the master tree code of length k (created by a pattern of $O(k)$ prior corruptions). Within this region the adversary can create inconsistencies that the parties will not be able to detect.

We use a recursive pattern of corruptions that is inserted by the adversary as follows. Suppose that for some parameters $l, E, C > 0$, there is a way for the invest E corruptions so as to make the parties need at least C communication to advance l steps in the unprotected region. We show that there is a way for the adversary to insert $2E+1$ corruptions that forces the parties to use at least $2C + l$ communication to advance $2l+1$ steps in the unprotected region. Unfolding the recursion, we get that there is a way for the adversary to insert $\Theta(k)$ corruptions so that the parties waste at least $\Omega(k \log k)$ communication to move out of the unprotected region of length k, as desired.

To see why, divide the unprotected region of length $2l + 1$ into three regions of lengths $l, 1, l$ respectively. The adversary uses E corruptions in the first l steps to make sure that the parties require at least C communication to move beyond l steps. As we assume that the patching tree codes are dropped at the point where the rewind process started, the parties drop all patching tree codes after l steps and still be in the unprotected region of the master tree code. Then, the adversary adds another *special* corruption in a way such it is only detected l steps later (this is possible as the the parties are in an unprotected region). During these l steps, the adversary uses E more corruptions to waste another C amount of communication by the parties.

Now, when the parties eventually detect this error after l communication, they will go back l places and the forward again in order to fix the special corruption. Thus, the total communication needed by the parties to advance $2l$ steps in the unprotected region is $2C + l$, as desired.