Maliciously Secure Massively Parallel Computation for All-but-One Corruptions

Rex Fernando[1], Yuval Gelles[2], Ilan Komargodski[3], and Elaine Shi[4]

[1] UCLA: rex1fernando@gmail.com.
[2] Hebrew University: yuval.gelles@mail.huji.ac.il.
[3] Hebrew University and NTT Research: ilank@cs.huji.ac.il.
[4] Carnegie Mellon University: runting@gmail.com.

Abstract. The Massive Parallel Computing (MPC) model gained wide adoption over the last decade. By now, it is widely accepted as the right model for capturing the commonly used programming paradigms (such as MapReduce, Hadoop, and Spark) that utilize parallel computation power to manipulate and analyze huge amounts of data.

Motivated by the need to perform large-scale data analytics in a privacy-preserving manner, several recent works have presented generic compilers that transform algorithms in the MPC model into secure counterparts, while preserving various efficiency parameters of the original algorithms. The first paper, due to Chan et al. (ITCS '20), focused on the honest majority setting. Later, Fernando et al. (TCC '20) considered the dishonest majority setting. The latter work presented a compiler that transforms generic MPC algorithms into ones which are secure against *semi-honest* attackers that may control all but one of the parties involved. The security of their resulting algorithm relied on the existence of a PKI and also on rather strong cryptographic assumptions: indistinguishability obfuscation and the circular security of certain LWE-based encryption systems.

In this work, we focus on the dishonest majority setting, following Fernando et al. In this setting, the known compilers do not achieve the standard security notion called *malicious* security, where attackers can arbitrarily deviate from the prescribed protocol. In fact, we show that unless very strong setup assumptions as made (such as a *programmable* random oracle), it is provably *impossible* to withstand malicious attackers due to the stringent requirements on space and round complexity.

As our main contribution, we complement the above negative result by designing the first general compiler for malicious attackers in the dishonest majority setting. The resulting protocols withstand all-but-one corruptions. Our compiler relies on a simple PKI and a (programmable) random oracle, and is proven secure assuming LWE and SNARKs. Interestingly, even with such strong assumptions, it is rather non-trivial to obtain a secure protocol.

1 Introduction

The Massively Parallel Computation (MPC[5]) model, first introduced by Karloff, Suri, and Vassilvitskii [45], is widely accepted as the *de facto* model of computation that abstracts modern distributed and parallel computation. This theoretical model is considered to best capture the computational power of numerous programming paradigms, such as MapReduce, Hadoop, and Spark, that have been developed to utilize parallel computation power to manipulate and analyze huge amounts of data.

In the MPC model, there is a huge data-set whose size is N. There are M machines connected via pairwise communication channels and each machine can only store $S = N^{\varepsilon}$ bits of information locally for some $\varepsilon \in (0, 1)$. We assume that $M \in \Omega(N^{1-\varepsilon})$ so that all machines can jointly at least store the entire data-set. This setting is believed to capture best large clusters of Random Access Machines (RAMs), each with a somewhat considerable amount of local memory and processing power, yet not enough to store the massive amount of available data. Such clusters are operated by large companies such as Google or Meta.

The primary metric of efficiency for algorithms in the MPC model is their *round complexity*. In general, the goal is to achieve algorithms which run in $o(\log_2 N)$ rounds; Ideally, we aim for algorithms with $O(1)$ or $O(\log \log N)$ rounds. The local computation time taken by each machine is essentially "for free" in this model. By now, there is an immensely rich algorithmic literature suggesting various non-trivial efficient algorithms for tasks of interest, including graph problems [1,2,3,5,6,7,8,10,15,14,26,29,39,34,48,49,56,60,43], clustering [11,13,31,38,62] and submodular function optimization [59,32,47,53].

Secure MPC. The MPC framework enables the algorithmic study of the large-scale data analytics commonly performed today. From a security point of view, a natural question is whether it is possible to do so in a privacy-preserving manner. This question is of increasing importance because numerous data analytics tasks we want to perform on these frameworks involve sensitive user data, e.g., users' behavior history on websites and/or social networks, financial transaction data, medical records, or genomic data. Traditional deployment of MPC is often centralized and typically hosted by a single company such as Google or Meta. However, for various reasons, it may not be desirable for users to disclose sensitive data in the clear to centralized cloud providers.

As a concrete motivating scenario, imagine that multiple hospitals each host their own patient records, but they would like to join forces and perform some clinical study on the combined records. In this case, each hospital contributes one or more machines to the MPC cluster, and the challenge here is how the hospitals can securely compute on their joint dataset without disclosing their patient records. In this scenario, since the hospitals are mutually distrustful, it is desirable to obtain a privacy guarantee similar to that of cryptographic secure multi-party computation. That is, we would like to ensure that *nothing*

[5] Throughout this paper, whenever the acronym MPC is used, it means "Massively Parallel Computation" and not "Multi-Party Computation".

is leaked beyond the output of the computation. More specifically, just like in cryptographic secure computation, we consider an adversary who can observe the communication patterns between the machines and also control some fraction of the machines. Note that all machines' outputs can also be in encrypted format such that only an authorized data analyst can decrypt the result; or alternatively, secret shared such that only the authorized data analyst can reconstruct. In these cases, the adversary should not be able to learn anything at all from the computation. We call MPC algorithms that satisfy the above guarantee *secure MPC*.

Why classical secure computation techniques fail. There is a long line of work on secure multiparty computation (starting with [41,16]), and so it is natural to wonder whether classical results can be directly applied or easily extended to the MPC model. Unfortunately, this is not the case, due to the space constraint imposed on each machine. Algorithms in the MPC model must work in as few rounds as possible while consuming small space. Note that since the number M of machines can be even larger than the space s of a machine, a single machine cannot even receive messages from all parties at once, since it would not be able to simultaneously store all such messages. This immediately makes many classical techniques unfit for the MPC model. In many classical works, a single party must store commitments or shares of all other parties' inputs [41,46,57,4,54,30]. Also, protocols that require simultaneously sending one broadcast message per party (e.g., Boyle et al. [20]) are unfit since this also implies that each party needs to receive and store a message from all other parties.[6]

State of the art. Chan, Chung, Lin, and Shi [25] put forward the challenge of designing secure MPC algorithms. Chan et al.'s main result is a compiler that takes any MPC algorithm and outputs a secure counterpart that defends against a malicious adversary who controls up to $1/3 - \eta$ fraction of machines (for a small constant η). The round overhead of their compiler is only a constant multiplicative factor, and the space required by each machine only increases by a multiplicative factor that is a fixed polynomial in the security parameter. *Malicious* security relies on the existence of a threshold FHE (TFHE) scheme, (simulation-extractable multi-string) NIZKs, and the existence of a common random string (CRS) that is chosen *after* the adversary commits to its corrupted set. If the protocol specification can be written as a shallow circuit, then a leveled TFHE scheme would suffice, and so the construction can be based on LWE [61,4,19]. Otherwise, we need a non-leveled scheme which we can get by relying on Gentry's bootstrapping technique [36]. This requires the TFHE scheme being "circular secure".[7]

[6] Some works design "communication preserving" secure computation protocols (for example, [55,50,44]) where the goal is to eliminate input/output-size dependency in communication complexity—all of these works only address the two parties setting.

[7] In a leveled scheme the key and ciphertext sizes grow with the depth of the circuit being evaluated. In contrast, in a non-leveled scheme these sizes depend only on the security parameter. Gentry's bootstrapping requires the assumption that ciphertexts

More recently, Fernando et al. [33] considered the *dishonest majority* setting and presented two compilers. The first compiler only applies to a limited set of MPC functionalities (ones with a "short" output) and the second applies to all MPC functionalities. Both their compilers rely on a public-key infrastructure (PKI) and they obtain security for a *semi-honest* attacker that controls all machines but one. The round overhead of their compilers is similarly only a constant multiplicative factor, and the space required by each machine only increases by a multiplicative factor that is a fixed polynomial in the security parameter. Their first compiler is secure assuming a TFHE scheme and the second compiler is secure assuming TFHE, LWE, and indistinguishability obfuscation [12,35]. Both compilers require the TFHE scheme to be "circular secure". This work leaves two very natural open questions:

- Is it possible to get malicious security in the dishonest majority setting? Here, no feasibility result is known under any assumption!
- Can we avoid non-standard assumptions in the dishonest majority setting?

1.1 Our Results

We make progress towards answering both of the above problems. Our contributions can be summarized as follows (with details following):

1. We prove that it is impossible to obtain a maliciously secure compiler for MPC protocols, no matter what computational assumptions are used. Our impossibility result works even if the compiler assumes a PKI, a common reference string, or even a non-programmable random oracle.
2. We complement the above impossibility result by presenting a maliciously secure compiler for MPC protocols, assuming a *programmable* random oracle, zero-knowledge SNARKs, and LWE. This result is our main technical contribution.
3. Lastly, we make a simple observation that allows us to get rid of the circular security assumption on TFHE that was made by Fernando et al. [33], as long as the protocol specification can be written as a shallow circuit. Thus, in this case, our observation can be used to re-derive [33]'s semi-honest long output protocol, relying only on LWE and indistinguishability obfuscation. This is useful since many MPC algorithms have very low depth, usually at most poly-logarithmic in the input size (e.g., [3,8,39,43] to name a few).

An impossibility result for semi-malicious compilers. We show an impossibility result for a generic compiler that results with a semi-malicious secure MPC. The impossibility result holds in the setting of Fernando et al. [33], where strong cryptographic assumptions as well as a PKI were used. In fact, the impossibility result shows that no matter what cryptographic hardness assumptions are used and even if the compiler relies on a PKI, a common reference string (CRS), or

remain semantically secure even when we use the encryption scheme to encrypt the secret decryption key.

a (non-programmable) random oracle, then no generic compiler can result with semi-malicious secure MPC protocols.

In more detail, we show that the restrictions imposed by the MPC model (the near constant round complexity along with the space constraint) make it impossible to implement certain functionalities in a (semi-malicious) secure manner. Specifically, we design a functionality for which there is a party whose outgoing communication complexity is roughly proportional to the number of parties. This, in turn, means that either the round blowup or the space blowup must be significant, leading to a contradiction. The functionality that we design assumes the existence of a one-way functions.

This impossibility result is inspired by a related lower bound due to Hubáček and Wichs [44] who showed that the communication complexity of any malicious secure function evaluation protocol must scale with the output size of the function. We extend their proof to the (multiparty, space constrained) MPC setting allowing various trusted setup assumptions.

A malicious compiler. We observe that the above impossibility result does not hold if the compiler relies on a *programmable* random oracle. To this end, as our second and more technical result, we give a compiler which takes as input any insecure MPC protocol and turns it into one that is secure against a *malicious* attacker that controls all machines but one. This compiler relies on a few assumptions: LWE, the existence of a programmable random oracle, and a *zero-knowledge succinct non-interactive arguments of knowledge* (zkSNARK). The compilation preserves the asymptotical round complexity of the original (insecure) MPC algorithm. This is the first secure MPC compiler for the malicious, all-but-one corruption setting, under any assumption.

Recall that a SNARK is a non-interactive argument system which is succinct and has a strong soundness guarantee. By succinct we mean that the proof size is very short, essentially independent of the computation time or the witness size. The strong soundness guarantee of SNARKs is knowledge-soundness that guarantees that an adversary cannot generate a new proof unless it knows a witness. This is formalized via the notion of an *extractor* which says that if an adversary manages to produce an acceptable proof, there must be an efficient extractor which is able to "extract" the witness. A SNARK is said to be *zero-knowledge* if the proof reveals "nothing" about the witness. There are many onstructions of SNARKs with various trade offs between efficiency, security guarantees, and the required assumptions, for example the works of [52,18,42]. All of these constructions can be used to instantiate our compiler, resulting in a compact CRS with size independent of the number of parties and the input and output size.

From semi-malicious to malicious. The above result is obtained in two steps. First, we obtain a *semi-malicious* MPC compiler. This step builds on the semi-honest (long output) protocol of Fernando et al. [33] and extends it to the semi-malicious setting. Second, we *generically* transform any semi-malicious MPC algorithm into a malicious one (both for all-but-one corruptions). The

transformation uses only zero-knowledge SNARKs and has only constant overhead in its round complexity.

We remark that our semi-malicious protocol does not need a random oracle if we only need semi-honest security. This result is interesting by itself since it gives a strict improvement over the result of Fernando et al. [33]. Indeed, we get the same result as that of [33] except that we use plain threshold FHE as opposed to their result which relies on a novel circular security assumption related to threshold FHE.

Recall that in semi-malicious security introduced by Asharov et al. [4], corrupt parties must follow the protocol specification, as in semi-honest security, but can use arbitrary values for their random coins. In fact, the adversary only needs to decide on the input and the random coins to use for each party in each round at the time that the party sends the first message.

Our semi-malicious to malicious compiler is essentially a GMW-type [41] compiler but for the MPC setting, and therefore is of independent interest. Interestingly, standard compilation techniques in the secure computation literature do not apply to the MPC model. For example, it is well-known that in the standard model, one can generically use non-interactive zero-knowledge proofs (NIZKs) to compile any semi-malicious protocol into a malicious one (see e.g., [4,54]) without adding any rounds. However, this transformation relies on a broadcast channel and is therefore inapplicable to the MPC model. We therefore present a relaxation of semi-malicious security, called *P2P semi-malicious security*, which fits better to a peer-to-peer communication network, and in particular, to the MPC model. We show a generic transformation from P2P semi-malicious security to malicious security, assuming LWE and a zkSNARK for NP. Our transformation is much more involved than the classical one in the broadcast model (which uses "only" zero-knowledge proofs) and requires us to design and combine several new primitives. We believe that this relaxation of semi-malicious security and the transformation themselves are of independent interest.

Paper Organization

In Section 2, we give an overview of the techniques used in obtaining our results. In Section 3 we formally defined the model and the malicious and P2P-semi-malicious security definitions. In Section 4 we prove the impossibility result of generic (semi-)malicious compilers even using setup assumptions. In Section 5 we introduce two commonly used procedures. In Section 6 we give a P2P-semi-malicious compiler for long-output MPC protocols. Lastly, in Section 7 we give our P2P-semi-malicious-to-malicious compiler.

2 Overview of our Techniques

First, let us briefly recall the computational model. The total input size contains N bits and there are about $M \approx N^{1-\varepsilon}$ machines, each having space $S = N^{\varepsilon}$. The space of each machine is bounded by S and so in particular, in each round

each machine can receive at most S bits. We are given some protocol in the MPC model that computes some functionality $f: (\{0,1\}^{l_{\mathsf{in}}})^M \to (\{0,1\}^{l_{\mathsf{out}}})^M$, where $l_{\mathsf{in}}, l_{\mathsf{out}} \leq S$, and we would like to compile it into a secure version that computes the same functionality. We would like to preserve the round complexity up to constant blowup, and to preserve the space complexity as much as possible. Ultimately, we want to guarantee the strong notion of security against malicious attackers that can arbitrarily deviate from the protocol specification.

Since our goal is to use cryptographic assumptions to achieve security for MPC protocols, we introduce an additional parameter λ, which is a security parameter. For a meaningful statement, one must assume that $N = N(\lambda)$ is a polynomial and that S is large enough to store $O(\lambda)$ bits.

We assume that the communication pattern, i.e., the number of messages sent by each party, the size of messages, and the recipients, do not leak anything about the parties' inputs. We call a protocol that achieves this property *communication oblivious*. This assumption can be made without loss of generality due to a result of Chan et al. [25] who showed that any MPC protocol can be made communication oblivious with constant blowup in rounds and space.

It is instructive to start by explaining where classical approaches to obtaining malicious security break down. A natural approach to bootstrap semi-honest to malicious security is by enforcing honest behavior. A semi-honest compiler was given by Fernando et al. [33] so this seems like a good starting point. Typically, such a transformation is done first letting parties commit to their (secret) inputs and running a coin-flipping protocol to choose randomness for all parties before the beginning of computations, and then together with every message they send, they attach a proof that the message is well formed and was computed correctly using the committed randomness. The proofs must be zero-knowledge so that no information is leaked about their input and randomness. This is the most common generic approach, introduced already in the original work of Goldreich, Micali, and Wigderson (GMW) [41]. It turns out that trying to adapt this approach to the MPC setting runs into many challenges.

- GMW-type compilers usually rely on a multiparty coin-flipping protocol since the underlying semi-honest protocol only guarantees security when parties use fresh and uniform randomness to generate their messages. It is not clear how to perform such a task *while respecting the constraints of the MPC model*.
- GMW-type compilers usually rely on an all-to-all communication pattern. That is, whenever a party P_i sends a message, it must prove individually to each other party P_j that it acted honestly. This, of course, is completely untenable in the MPC model, since it would mean an $O(M)$ blowup in communication. Specifically, assume party P_1 sends a message m during round 1. Even if the message is meant only for P_2, the GMW compiler requires P_1 to broadcast a commitment c_m of m to every other party too, and prove that the message committed under c_m is computed correctly. This is necessary because other parties may later on receive messages from, say, P_2, that depend on m. This approach incurs $O(M)$ communication blowup,

and this blowup must be charged either to round complexity or space in the MPC model.

The impossibility result. It turns out that the above challenges are somewhat inherent in the MPC setting in the sense that *it is impossible to bypass them*, even if arbitrarily strong cryptographic assumptions are made or even if trusted setup assumptions are used (e.g., a PKI or a non-programmable random oracle). Specifically, it is impossible to obtain a maliciously-secure MPC compiler under any cryptographic assumption and even if various setup assumptions are used.

The main idea for this impossibility result is to consider the following functionality: party 1 holds as input a PRF key k and the functionality is to send to party $i \in [M]$ the value of the PRF at point i, i.e., $\mathsf{PRF}_k(i)$. The attacker will control all parties but 1. We show that any malicious compiler for this functionality must incur non-trivial overhead either in the round complexity or in the space complexity, rendering it useless in the MPC setting. More specifically, we show that the total size of outgoing communication from party 1 must be proportional to the number of machines in the system, M, which in turn means it must store this many bits in a small number of rounds, implying our result. The proof of this lower bound on the outgoing communication complexity of party 1 is inspired by a related lower bound due to Hubáček and Wichs [44] who showed that the communication complexity of any malicious secure function evaluation protocol (a 2-party functionality) must scale with the output size of the function. We extend their proof to the (multiparty, space constrained) MPC setting and also to capture various trusted setup assumptions.

The main idea of the proof is as follows. The view of the adversary in any realization of the above protocol contains about M outputs of the PRF. By security, these outputs should be efficiently simulatable. If the communication complexity from party 1 is smaller than M, we can use the simulator to efficiently compress about M PRF values. This contradicts the fact that the outputs of a PRF are incompressible. The actual argument captures protocols that might rely on setup which is chosen before the inputs, for instance, a PKI or a non-programmable random oracle. Additionally, the above argument works even if the underlying MPC is not maliciously secure but only "semi-malicious" or even our new notion "P2P-semi-malicious" (both of which will be discussed below).

2.1 Our Malicious Compiler for Short Output Protocols

To explain the main ideas underlying our compiler we first focus on a simpler setting where the given (insecure) MPC algorithm has an output that fits into the memory of a single machine. Following the terminology of Fernando et al. [33], we call such protocols *short output*. Recall that our impossibility result from above basically says that the outgoing communication complexity from some party must scale with the total output size. Since the latter is very small in our case, we conclude that the impossibility result does not apply to short output MPC protocols.

Our starting point is the semi-honest compiler of Fernando et al. [33]: execute Π under the hood of a homomorphic encryption (HE) scheme and eventually (somehow) decrypt the result. If implemented correctly, intuitively, it is plausible that such a protocol will guarantee security for any single party, even if all other ones are colluding. The main question is basically how to decrypt the result of the computation. Fernando et al. [33] relied on a threshold FHE scheme to implement the above blueprint.

At this point we would like to emphasize that it is not immediately straightforward how to adapt existing threshold- or multi-key-based FHE [4,51,54,22,58,19,9] solutions to the MPC model. At a high-level, using these tools, each party first broadcasts an encryption of its input. Then each party locally (homomorphically) computes the desired function over the combined inputs of all parties, and finally all parties participate in a joint decryption protocol that allows them to decrypt the output (and nothing else). However, the classical joint decryption protocols are completely non-interactive but consume high space: each party broadcasts a "partial decryption" value so that each party who holds partial decryptions from all other parties can locally decode the final output of the protocol. If the underlying MPC protocol is short output, then we can leverage the fact that for known TFHE schemes, the joint decryption process can be executed "incrementally" over a tree-like structure, making it perfectly fit into the MPC model. Specifically, it is possible to perform a joint decryption protocol in the MPC model to recover the output *as long as it fits into the memory of a single machine*.

Avoiding Coin-Flipping (or: P2P Semi-Malicious Security). As mentioned, we do not know how to directly perform a multiparty coin-flipping protocol in the MPC model. Many previous works, such as [4,54], bypass this problem in the name of saving rounds of communication, by assuming that the underlying protocol satisfies a stronger notion of security called *semi-malicious* security.

In semi-malicious security, the guarantee is similar to semi-honest security, namely, that the attacker has to follow the protocol, except that it is free to choose its own randomness. This is formalized by the requirement that after every message the adversary sends on behalf of a corrupted party, it must *explain* all messages sent up to this point by the party by providing an input and randomness which is consistent with this party's messages.

We do not know if the above semi-honest protocol can be proven to satisfy semi-malicious security. This is because the classical definition of semi-malicious security seems specifically defined to work in the broadcast model, and there are subtle problems that arise when using it without broadcast. Nevertheless, we manage to define a relaxation of semi-malicious security, we term *P2P semi-malicious security*, which turns out to be easier to work with in the MPC model. With this refine notion in hand, we show that the above-mentioned semi-honest MPC compiler satisfies P2P semi-malicious security. This step is rather straightforward once the right definition is in place. The main technical

contribution is a method to bootstrap this (weaker) notion of security to full-fledged malicious security.

To explain what P2P semi-malicious security is, it is instructive to be more precise about what semi-malicious security means. Specifically, in the semi-malicious corruption model the adversary is only required to give a local explanation of each corrupted party's messages. In the broadcast channel, this is not a problem because all messages are public anyway, even those *between* corrupted parties. However, absent a broadcast channel, the adversary need not explain messages between corrupt parties as they can essentially be performed by the attacker, outside of the communication model. (Recall that in the definition of secure computation, an adversary is only required to furnish messages which honest parties can see.) Thus, in the P2P semi-malicious security model, we require the adversary to explain its behavior completely by also explaining the "hidden" messages sent amongst corrupt parties. While this gives a weaker security guarantee than classical semi-malicious security it is still stronger than semi-honest security and it turns out to be sufficient for us to go all the way to malicious security.

Enforcing P2P Semi-Malicious Behavior The next challenge is how to compile our P2P semi-malicious protocol into a maliciously secure one. Recall that classical GMW-type [41] compilers do not work in the MPC model since whenever a party P_i sends a message, it must prove individually to each other party P_j that it acted honestly. This, of course, is completely untenable in the MPC model, since it would mean an $O(M)$ blowup in communication. Specifically, assume party P_1 sends a message m during round 1. Even if the message is meant only for P_2, the GMW compiler requires P_1 to broadcast a commitment c_m of m to every other party too, and prove that the message committed under c_m is computed correctly. This is necessary because other parties may later on receive messages from, say, P_2, that depend on m. This approach incurs $O(M)$ communication blowup, and this blowup must be charged either to round complexity or space in the MPC model.

First attempt. We use a strong form of zero-knowledge proofs, known as *succinct non-interactive arguments of knowledge* or zkSNARKs. These proofs have the useful property that they can be recursively composed without blowing up the size of the proofs. What this means is that if a verifier sees a proof π for some statement x, it can then compute a new proof π' that attests to knowledge of a valid proof π for x. In our setting, this means that every party P_i can prove that its (committed) new state and outgoing messages are computed correctly based on its committed previous state and random coins, as well as a set of incoming messages which themselves must carry valid zkSNARKs that vouch for their validity.

Remark 1. Note that for this to work in the MPC model, we need that the proofs are computable in space proportional to the space of the local round computation. A SNARK that preserves the space bound in this way is called a

complexity-preserving SNARK, and generic transformations from any SNARK to a complexity-preserving one are known [18].

Unfortunately, proving the security of this scheme turns out to be problematic. In the security proof, we would need to recursively extract the composed zkSNARK proofs in order to find the "cheating" proof (as some proofs along the way could be correct). That is, we need to invoke the SNARK extractor over an adversary *that itself is an extractor*. Unfortunately, performing this recursive extraction naïvely blows up the running time of the extractor *exponentially* with the depth of the recursion, and thus the recursive composition can only be performed $O(1)$ times. This means that we would be able to support only constant-round MPC protocols. Some works bypass this problem by making the very strong, non-standard assumption that there is a highly efficient extractor (i.e., where the overhead is *additive*) and therefore recursive composition can be performed for an unbounded number of times (for example, [18,28,17,21]). We want to avoid such strong assumptions.

Remark 2. At a very high level, proofs carrying data (PCDs) [27,18,24,23] are a generalization of classical proofs that allow succinctly proving honest behavior over a distributed computation graph. While the communication underlying an MPC algorithm can be viewed as a specific distributed computation, we cannot use them directly to get malicious security. The main problem is that PCDs only exist for restricted classes of graphs, unless very strong assumptions are made. Known PCDs (e.g., the one of Bitansky et al. [18] which in turn relies on SNARKs) only support constant-depth graphs or polynomial-depth paths. Our protocol does not fit into either of these patterns. It is possible to get PCD for arbitrary graphs from SNARKs where the extractor has an *additive polynomial-time overhead*, but as mentioned we want to avoid such strong assumptions. Second, note that we require privacy when compiling a malicious-secure protocol, which PCDs alone do not guarantee.

Our solution. Our goal is however to support an arbitrary round MPC protocol. To this end, we devise a method for verifying P2P-semi-malicious behavior by ensuring consistency and synchrony of intermediate states after every round, instead of throughout the whole protocol. We want that at the end of each round, the parties collectively hold a succinct commitment to the entire current state of all parties in the protocol. Given this commitment and a commitment to the previous round's state, the parties then collectively compute a proof that the entire current-round state has been obtained via an honest execution of one round of the protocol with respect to the previous-round state. This is implemented by recursively composing succinct proofs about the *local* state of each machine (using its limited view of the protocol execution) into a conjunction of these statements which proves *global* honest behavior. We implement this sub-protocol by composing zkSNARKs in a tree-like manner so that we only have constant blow up in round complexity.

To describe our approach, we first design a few useful subprotocols:

1. CalcMerkleTree sub-protocol: every party $i \in [M]$ has an input x_i, and they run an MPC protocol such that everyone learns the root digest τ of a Merkle tree over $\{x_i\}_{i\in[M]}$, and moreover, every party learns an opening that vouches for its own input x_i w.r.t. to the Merkle root τ. We make the arity γ of the Merkle tree as large as λ, i.e., the security parameter, and therefore the depth of the tree is a constant. The protocol works in the most natural manner by aggregating the hash over the γ-fan-in Merkle tree: in every level of the tree, each group of γ parties send their current hash to a designated party acting as the parent; and the parent aggregates the hashes into a new hash. At this moment, we can propagate the opening to each party, this time in the reverse direction: from the root to all leaves. The protocol completes in constant number of rounds.
2. Agree sub-protocol: every party $i \in [M]$ has an input x_i, and they run a secure MPC protocol to decide if all of them have the same input. To accomplish this in constant number of rounds in the MPC model without blowing up the space, we use a special threshold signature scheme that allows for distributed reconstruction. We build such a signature scheme by adapting a scheme of Boneh et al. [19] to our setting. Crucially, the signature scheme of [19] has a reconstruction procedure which is essentially *linear*. In this way, the parties can aggregate their signature shares over a wide-arity, constant-depth tree (where the arity is again λ). If all parties do not have the same input, disagreement can be detected during the protocol. Otherwise, the party representing the root obtains a final aggregated signature which is succinct. It then propagates the signature in the reverse direction over the tree to all parties.
3. RecCompAndVerify subprotocol: every party $i \in [M]$ holds a zkSNARK proof π_i to some statement stmt_i. They run an MPC protocol to compute a recursively composed proof π for a statement that is the conjunction of all stmt_is. This is also performed by aggregating the proofs over a wide-arity, constant-depth tree (where the arity is again λ); and the aggregation function in this case is the recursive composition function of the zkSNARK. The aggregated proof is propagated in the reverse direction over the tree to all parties.

In the first phase, before the protocol begins, each party holds an input and randomness for the underlying protocol which is being compiled. First, the parties engage in a "commitment phase": (1) each party computes a non-interactive hiding and binding commitment to their input and randomness, and then (2) the parties collectively generate a Merkle root τ_0 which commits to these commitments. This step can be accomplished by 1) calling CalcMerkleTree subprotocol, at the end of which every party receives a Merkle root τ_0 and its own opening, and 2) calling Agree to ensure everyone agrees on τ_0.

The next phase is used to simulate the first round of the underlying protocol. Recall that at this point all parties have a consistent Merkle root τ_0 which commits to their inputs and randomnesses, so in other words, τ_0 commits to the global starting state of the protocol. Each party executes the underlying protocol to obtain a new private state $\mathsf{st}_{i,1}$ and a list $\mathsf{msg}^{\mathsf{out}}_{i,1}$ of its outgoing messages. It then sends these outgoing messages to the recipient parties, and also stores

any messages it received in a list $\mathsf{msg}^{\mathsf{in}}_{i,1}$. Every party now has a combined local state $(\mathsf{st}_{i,1}, \mathsf{msg}^{\mathsf{in}}_{i,1}, \mathsf{msg}^{\mathsf{out}}_{i,1})$. All parties now collectively compute a Merkle root τ_1 of all their combined states, again by calling the $\mathsf{CalcMerkleTree}$ and Agree sub-protocols. The parties now need to compute a new succinct proof π_1 that the entire round-1 state committed to by τ_1 has been honestly computed with respect to τ_0. For this to be true, each party P_i must not only prove that $(\mathsf{st}_{i,1}, \mathsf{msg}^{\mathsf{out}}_{i,1})$ was honestly computed, but also that its outgoing messages have been properly received by its recipients. At this point, each party P_i replies to every party P_j that sent a message that it received with an opening in the global state τ_1 that proves that it has recorded the message correctly in its list $\mathsf{msg}^{\mathsf{in}}_{i,1}$. Now, each party P_i can compute a proof that $(\mathsf{st}_{i,1}, \mathsf{msg}^{\mathsf{out}}_{i,1})$ has been computed honestly, and in addition, that every message in $\mathsf{msg}^{\mathsf{out}}_{i,1}$ has been copied to $\mathsf{msg}_{j,1}$, where P_j is the recipient of the message. These proofs are again aggregated using a recursive composition tree into a single succinct proof, by calling $\mathsf{RecCompAndVerify}$.

The rounds which follow proceed in essentially the same way. The only difference is that in successive rounds, the input to each party's local computation also includes the incoming messages $\mathsf{msg}^{\mathsf{in}}_{i,r}$. In this way, the parties incrementally verify that the protocol is being performed honestly, without ever having to store a full transcript of the execution.

In terms of efficiency, the above compiler essentially replaces each round of the underlying protocol with a constant number of rounds, therefore the round blowup is constant. Moreover, the extra local space per party needed to carry out this transformation is only $\mathsf{poly}(\lambda) \cdot S$, where λ is the security parameter and S is the space bound of the underlying protocol.

Technicalities in the proof. The most interesting challenge that arises is handling the recursive composition of the SNARKs. In particular, we are in the context of secure computation, and therefore we are required to exhibit a simulator which can replicate the behavior of an adversary in the ideal world. This means we will need to simulate the honest parties' SNARKs, so we will require SNARKs with a zero-knowledge property, or zkSNARKs. Moreover, we need to extract the corrupted parties' witnesses. Since the recursive composition tree in the simulated world will include a combination of real and simulated proofs, it is not clear how to use a standard SNARK extractor to extract in this setting. To overcome this, we use a stronger notion of extraction, known as identity-based simulation extractability, which works even in the presence of simulated proofs [21]. At a high level, in this type of SNARK, each party receives an identity and proves statements with respect to that identity. Then, during extraction, the adversary receives a restricted trapdoor which allows it to simulate any proof with an honest id. The extractor is then guaranteed to extract valid witness for any proof generated with an id that is not in the honest set. Crucially, this notion is implied by the existence of vanilla SNARKs for NP along with one-way functions, as shown by Boyle et al. [21]. (Note that the transformation of [18] for complexity preserving SNARKs also preserves the identity-based simulation extractability property.)

Using the simulation-extractability property of a SNARK in the context of secure computation protocols is trickier than the analogous usage of (non-succinct) non-interactive arguments. In particular, the extractor needs to make non-black-box use of the adversary [37]. A naive way for the simulator to use this property would be to extract the witnesses used by the corrupted parties in every round, and then to verify using the witnesses that the round was computed honestly. Unfortunately, it is not clear how to run the extractor in every round without recursively composing the extractor with itself R times. This is problematic in super-constant round protocols, because the extraction time could depend double-exponentially in the number of rounds. This appears to be even more of a problem for the following reason. Since we are using recursively composable SNARKs, soundness of our proofs are only guaranteed by exhibiting an extractor, and thus it seems like extraction in every round is inevitable. However, we bypass this problem by forcing the corrupted parties to commit at the very beginning to all randomness which they use in the protocol, even the randomness used to commit to their private state in each round. This allows us to write a simulator which only extracts in the first "commit phase" round, and also allows us to guarantee soundness via a reduction which only extracts in the first round and some other arbitrarily chosen round (the reduction is to the collision-resistance of the hash function or the binding property of the commitment scheme). See details in Appendix 7.

2.2 Our Malicious Security for Long Output Protocols

Now, we consider general MPC protocols. Due to our impossibility result, obtaining an analogous result for general MPC protocols necessarily requires a new approach, even if we only want to get (P2P) semi-malicious security. To put things in context, it is useful to recall the *semi-honest* MPC compiler for general MPC protocols of Fernando et al. [33].

Recall that the main challenge is to perform joint decryption of threshold FHE ciphertexts where each party eventually wants to learn its own output. Here is where Fernando et al. [33] used indistinguishability obfuscation: they generate an obfuscated circuit that has the master secret key hardwired and only agrees to decrypt the given M ciphertexts. Ensuring that this circuit itself is succinct requires careful use of SSB hashing [44] among other techniques. Once the circuit is small enough, they invoked their short output protocol to generate it securely and then distribute to all parties.[8]

This MPC compiler provably (due to our impossibility result) does *not* result in P2P-semi-malicious MPC protocols. Moreover, it is not a matter of throwing in more cryptographic assumptions or modifying the protocol in some clever way—any such modification will still result with an insecure protocol against semi-malicious attackers. Our main observation used to bypass this is that the

[8] Recall that no party knows the master secret key and so an inner short-output protocol is executed. Its inputs include the shares of the master secret key and it outputs an obfuscation of the aforementioned circuit.

impossibility result fails for protocols where the simulator can program the setup *adaptively, depending on the private inputs of the parties.* To this end, we rely on a programmable random oracle to "program" a specific uniformly-looking value, tying the hands of semi-malicious attackers.

In more detail, at the end of the evaluation phase, each party holds an encryption of its output. These outputs are (homomorphically) padded, and then all of these padded ciphertexts are used in a joint protocol to compute a "restricted decryption" obfuscated circuit. Additional randomness is generated by each party by querying the random oracle and is hardwired (in a hashed manner) in the restricted decryption circuit; this randomness is generally ignored throughout the protocol. The simulator will use these random values to program the "right" values to be output by the restricted decryption circuit. Specifically, in semi-malicious security, after each party commits to its input and randomness, the simulator knows the private inputs and pads of malicious parties. At this point, it can program the random oracle at the appropriate location so that using it to mask the padded output gives the right output. We refer to Section 6 for the precise details.

From semi-malicious to malicious. To compile the above semi-malicious protocol into a malicious one, we essentially use the same compiler that we described in Section 2.1. Indeed, that compiler did not rely on the underlying MPC being short output at any point. The only technical issue is that we need to address the fact that the underlying semi-malicious MPC compiler uses a random oracle which makes it delicate in combination with SNARKs whose goal is to enforce honest behaviour. To overcome this problem, we carefully design the semi-malicious protocol in a way that allows us to separate the random oracle-related computation from the statement that is being proven via the SNARKs. Specifically, we design the semi-malicious MPC compiler so that the "important" points of the random oracle are known to all parties and so parties can locally verify that part of the computation without using a SNARK, and the SNARK will only apply to the other part of the computation which is in the plain model.

3 The MPC Model and Security Definitions

In this section we formally define the MPC model and then define appropriate security definitions. The model is defined in Section 3.1. The security of MPC algorithms is defined in a standard way, following the security definition in multi-party computation literature. We focus on the strongest notion of security called *malicious* security but we also define a weaker notion called *semi-malicious* security which we use as a stepping stone towards malicious security (see Sections 3.2 and 3.3, respectively).

3.1 The Massively Parallel Computation Model

We briefly recall the massively parallel computation (MPC) model, following Chan et al. [25] and refer to their work for a more detailed description. In the

MPC model, there are M parties (also called machines) and each party has a local space of S bits. The input is assumed to be distributively stored in each party, and let N denote the total input size in bits. It is standard to assume $M \geq N^{1-\varepsilon}$ and $S = N^{\varepsilon}$ for some small constant $\varepsilon \in (0, 1)$. Note that the total space is $M \cdot S$ which is large enough to store the input (since $M \cdot S \geq N$), but at the same time it is not desirable to "waste" space and so it is commonly further assumed that $M \cdot S \in \tilde{O}(N)$ or $M \cdot S = N^{1+\theta}$ for some small constant $\theta \in (0, 1)$. Further, assume that $S = \Omega(\log M)$.

At the beginning of a protocol, each party receives an input, and the protocol proceeds in rounds. During each round, each party performs some local computation given its current state, and afterwards may send messages to some other parties through private authenticated pairwise channels. An MPC protocol must respect the space restriction throughout its execution—namely, each party may store at any point in time during the execution of the protocol at most S bits of information (which in turn implies that each party can send or receive at most S bits in each round). When the protocol terminates, the result of the computation is written down by all machines to some designated output tape, and the output of the protocol is interpreted as the concatenation of the outputs of all machines. In particular, an output of a given machine is restricted to at most S bits. An MPC algorithm may be randomized, in which case every machine has a sequential-access random tape and can read random coins from the random tape. The size of this random tape is not charged to the machine's space consumption.

In this paper, we will be compiling MPC algorithms into secure counterparts and so it will be will be convenient to make several assumptions about the underlying (insecure) MPC, denoted Π:

- In protocol Π, each party P_i takes a string x_i of size l_{in} as input and outputs a string y_i of size l_{out}, where $l_{\mathsf{in}}, l_{\mathsf{out}} \leq S$. It follows that $N = l_{\mathsf{in}} \cdot M$.
- Let R be the number of rounds that the protocol takes. In each round $r \in [R]$, the behavior of party $i \in [M]$ is described as a circuit $\mathsf{NextSt}_{i,r}$. We assume that $\mathsf{NextSt}_{i,r}$ takes a string $\mathsf{st}_{i,r-1}||\mathsf{msg}^{\mathsf{in}}_{i,r-1}$ as an input and outputs string $\mathsf{st}_{i,r}||\mathsf{msg}^{\mathsf{out}}_{i,r}$, where $\mathsf{st}_{i,r}$ is the *state* of party i in round r and $\mathsf{msg}^{\mathsf{in}}_{i,r-1}$, the incoming messages to party i in round $r-1$, and $\mathsf{msg}^{\mathsf{out}}_{i,r}$ are the outgoing messages of party i in round r. Note that the space of each party is limited to S bits, so in particular $|\mathsf{st}_{i,r}| \leq S$ for each $i \in [M]$ and $r \in [R]$.
- The protocol is *communication-oblivious*: in round $r \in [R]$, each party P_i sends messages of a prescribed size to prescribed parties. In particular, this means that the communication pattern of the protocol is independent of the input and therefore does not leak any information about it. This assumption is without loss of generality due to a transformation from the work of Chan et al. [25] who showed that any MPC protocol can be transformed into a communication-oblivious one with only a constant multiplicative factor in the number of rounds.

3.2 Malicious Security for MPC protocols

We now define malicious security for MPC protocols following the general real-ideal framework (given e.g., in [40]) for defining secure protocols. We consider protocols assuming a PKI and a random oracle. Security is shown by exhibiting a simulator which can generate a view that is indistinguishable from the adversary's real-world view. We want to handle adversaries which can cause the corrupted parties to deviate arbitrarily from the protocol specification. To do that, we define real-world and ideal-world executions as follows. We consider an MPC protocol Π which realizes a functionality $f(x_1, \ldots, x_M) \to (y_1, \ldots, y_M)$.

Communication model and setup. Our protocols will assume authenticated pairwise channels between parties, such that a message sent from an honest party P_i to an honest party P_j is always received by P_j at the end of the round in which it was sent. We assume that the adversary can see all messages sent between honest parties. On the other hand, we do not assume honest parties can see messages between corrupted parties. We note that since our security definition will allow aborts, it is not necessary to prevent "flooding" attacks.

Furthermore, our protocol will rely on trusted setup, i.e., a PKI and a random oracle. The public key of the PKI is denoted pk and the random oracle is denoted $\mathcal{O}$. Every party in the protocol (including the adversary) has query access to $\mathcal{O}$.

The real-world execution. In the real-world execution, the protocol Π is carried out among the M parties, where some subset $\mathcal{C}$ of corrupted parties is controlled by the adversary $\mathcal{A}$. $\mathsf{real}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ a random variable whose value is the output of the execution which is described as follows. First, $\mathcal{A}$ is initialized with security parameter 1^λ. $\mathcal{A}$ first chooses an input size (the length of $x_1||\ldots||x_M$). After receiving the public key pk and the number M of parties, $\mathcal{A}$ chooses a set $\mathcal{C}$, and then receives the set $\{(x_i, \mathsf{sk}_i)\}_{i\in\mathcal{C}}$ of the corrupted parties' inputs and secret keys. The honest parties are then initialized with the inputs $\{x_i\}_{i\in[M]\setminus\mathcal{C}}$, and then $\mathcal{A}$ performs an execution of Π with the honest parties, providing all messages on behalf of the corrupted parties. Note that $\mathcal{A}$ does not need to provide messages sent between corrupted parties, since the honest parties do not see these messages. At the end of the protocol execution, $\mathcal{A}$ may output an arbitrary function of its view. Note that throughout the experiment $\mathcal{A}$ may perform arbitrary queries to the oracle $\mathcal{O}$. The output of $\mathsf{real}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ is defined to be a tuple consisting of the output of $\mathcal{A}$, a sequence of input-output pairs corresponding to the oracle queries that were made, along with the outputs of all honest parties.

The ideal-world execution with abort. The ideal-world execution is given with respect to the function f which is computed by an honest execution of Π. In the ideal world, an adversary $\mathcal{S}$, called the simulator, interacts with an ideal functionality $\mathcal{F}^f$. Denote with $\mathsf{ideal}^{\mathcal{F}^f}_{\mathcal{S}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ the output of the execution which is defined as follows:

- **Choosing input size and the corrupted set:** First, $\mathcal{S}$ chooses an input size. After receiving M, $\mathcal{S}$ chooses the set $\mathcal{C}$ of corrupted parties, and receives the set $\{x_i\}_{i\in\mathcal{C}}$. The honest parties $\{P_i\}_{i\in[M]\setminus\mathcal{C}}$, are each initialized with input x_i.
- **Sending inputs to the trusted party:** Every honest party sends its input x_i to $\mathcal{F}^f$, and $\mathcal{F}^f$ records $\tilde{x}_i = x_i$. $\mathcal{S}$ sends a set $\{\tilde{x}_i\}_{i\in\mathcal{C}}$ of arbitrary inputs, where each $\tilde{x}_i$ is not necessarily equal to x_i.
- **Trusted party sends the corrupted parties' outputs to the adversary:** $\mathcal{F}^f$ now computes $f(\tilde{x}_1, \ldots \tilde{x}_M) \to (y_1, \ldots, y_M)$. It sends $\{y_i\}_{i\in\mathcal{C}}$ to $\mathcal{S}$
- **Adversary chooses which honest parties will abort:** $\mathcal{S}$ now sends the set $\{\mathsf{instr}_i\}_{i\in[M]\setminus\mathcal{C}}$ to $\mathcal{F}^f$, where for each i, instr_i is either "continue" or "abort". $\mathcal{F}^f$ then sends output y_i to each honest party P_i where instr_i is "continue", and sends output $\perp$ to each honest party P_i where instr_i is "abort".
- **Outputs:** $\mathcal{S}$ outputs an arbitrary function of its view. The output of the execution is defined to be a tuple consisting of $\mathcal{S}$'s output along with all outputs of the honest parties.

We now define malicious security for MPC protocols formally in terms of the real-world and ideal-world executions.

Definition 1. *We say that an MPC protocol* Π *for a functionality* f *is* malicious secure in the PKI model and random oracle model *if for every non-uniform polynomial-time adversary* $\mathcal{A}$ *there exists a non-uniform polynomial-time simulator* $\mathcal{S}$ *such that for every ensemble* $\{x_i\}_{i\in[M]}$ *of poly-size inputs,* $\mathsf{real}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ *is computationally indistinguishable from* $\mathsf{ideal}^{\mathcal{F}^f}_{\mathcal{S}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$.

Remark 3 (Programmability). The above definition allows the simulator $\mathcal{S}$ to "program" the random oracle answers in its simulation. To allow this, the distinguisher in Definition 1 must not have access to this oracle.

Alternatively, sometimes a non-programmable variant is used. Specifically, here the distinguisher in Definition 1 *does* have access to this oracle and so $\mathcal{S}$ cannot program answers to its choice.

3.3 P2P Semi-Malicious Security for MPC protocols

We define a variant of semi-malicious security which is designed to be more suitable for models other than the broadcast model. We first explain why the original semi-malicious definition yields subtle problems when we do not assume the existence of a broadcast channel, and then we describe our modification to the definition.

In the original definition of semi-malicious security given by [4], a semi-malicious adversary is only required to give a *local* explanation of its behavior. Namely, whenever the adversary sends a message on behalf of a corrupted party P_i, the adversary must write to the witness tape an input-randomness pair for

P_i. It must be the case that the message just sent by P_i, along with all previous messages sent from P_i, are consistent with the input and randomness given by the adversary. Note that the adversary gives such an input-randomness pair whenever any corrupted party sends a message that is visible to honest party.

If we assume all communication takes place via a broadcast channel, this means that all messages are visible to honest parties, even messages between corrupted parties. Thus, an adversary is restricted to following the protocol specification honestly, with the proviso that it can change the input/randomness pairs for the corrupted parties partway through the protocol.

In the case of point-to-point channels, adversary is not required to furnish messages between corrupted parties, because they are not assumed to be visible to the honest parties. So although the adversary must explain any message sent from a corrupted party P_i to an honest party P_j with an input/randomness pair which is consistent with P_i's message, it is not required to explain the messages which were received by P_i from other corrupted parties. Since it can lie about the messages received by P_i from corrupted parties in previous rounds, the adversary can behave very differently from the honest protocol behavior. Thus, point-to-point channels offer much more freedom to a semi-malicious adversary than the standard case of a broadcast channel.

Our variant of this definition is designed to fix this problem and to bring the adversary's behavior back to what semi-malicious security is intuitively supposed to guarantee, namely that the adversary must act according to the honest protocol specification, modulo choosing the randomness for the corrupted parties and choosing different input/randomness pairs in different rounds.

We define our variant of semi-malicious security, which we call *security against P2P semi-malicious adversaries*, or *P2P semi-malicious security* for short. Like in malicious security definition, we use the real-ideal paradigm, the "semi-malicious" real-world execution is defined below, and the ideal-world execution is the same as in the malicious security definition from Section 3.2. Again, as before, we consider protocols in the PKI model and also in the presence of a random oracle that all participating parties (including the adversary) can query at any point in time. (Note that Remark 3 about programmability of the random oracle applies here as well.)

The real-world execution. In the real-world execution, the protocol Π is carried out among the M parties, where some subset $\mathcal{C}$ of corrupted parties is controlled by a *P2P semi-malicious adversary* $\mathcal{A}$. Denote $\mathsf{smReal}^{\Pi}_{\mathcal{A}}(1^{\lambda}, 1^{M}, \{x_i\}_{i\in[M]})$ a random variable whose value is the output of the execution which is described as follows. The real-world execution is similar to the real-world execution in the case of malicious security, except that we restrict the set of adversaries to P2P-semi-malicious ones. Such an adversary is required to have a special output tape called the "witness tape", and after each round ℓ it must write explanation of it behavior to this tape. That is, the adversary must write to the witness tape a set $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ consisting of an input and randomness for *every* corrupted party. (This is in contrast to standard semi-malicious security, where the adversary need only write the input and randomness (x_j, r_j) of each

corrupted party which sent a message to an honest party.) Observe that the messages sent by any party in $\mathcal{C}$ in the honest protocol specification up to and including round ℓ are uniquely determined by $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ and the setup (PKI and random oracle queries determined by the honest protocol specification) along with all messages sent from $[M] \setminus \mathcal{C}$ to $\mathcal{C}$ in previous rounds. Note that the witnesses given in different rounds not need be consistent. Also, we assume that the attacker is rushing and hence may choose the corrupted messages and the witness $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ in each round adaptively, after seeing the protocol messages of the honest parties in that round. Lastly, the adversary may also choose to abort the execution on behalf of $\{P_i\}_{i\in\mathcal{C}}$ in any step of the interaction. At the end of the protocol execution, $\mathcal{A}$ may output an arbitrary function of its view. Note that throughout the experiment $\mathcal{A}$ may perform arbitrary queries to the oracle $\mathcal{O}$. The output of $\mathsf{smReal}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ is defined to be a tuple consisting of the output of $\mathcal{A}$, a sequence of input-output pairs corresponding to the oracle queries that were made, along with the outputs of all honest parties.

Definition 2. *We say that an MPC protocol* Π *for a functionality* f *is* P2P semi-malicious secure in the PKI model and random oracle model *if for every non-uniform polynomial-time P2P semi-malicious adversary* $\mathcal{A}$ *there exists a non-uniform polynomial-time* $\mathcal{S}$ *such that for every ensemble* $\{x_i\}_{i\in[M]}$ *of poly-size inputs,* $\mathsf{smReal}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ *is computationally indistinguishable from* $\mathsf{ideal}^{\mathcal{F}^f}_{\mathcal{S}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$.

4 Impossibility of a (Semi-)Malicious Secure Compiler

In this section we prove that there is no generic compiler from insecure MPC protocol to semi-malicious secure counterparts. Our impossibility works even in the presence of various setup models. For instance, even if there is a PKI, a common reference string, and a (non-programmable) oracle, our result rules out a generic compiler.

Theorem 4.1. *Assume that there is a pseudorandom function family (PRF). Then, there is no generic compiler that takes as input an MPC protocol and outputs a P2P-semi-malicious MPC protocol that realizes the same functionality, unless the round complexity depends polynomially on the number of machines. This is true even if the compiler relies on a PKI or a (non-programmable) random oracle.*

Overview. The proof relies on the fact that a too-good-to-be-true compiler could be used to efficiently "compress" the outputs of a PRF. This is inspired by a result of Hubáček and Wichs [44] who showed that the communication complexity of any malicious secure function evaluation protocol must scale with the output size of the function. We extend their proof to the (multiparty, space constrained) multi party computation setting and also to capture various trusted setup assumptions.

More specifically, the hard functionality is one where party P_1 has, as input, a PRF key k and it wants to transmit the value of the PRF at location

$i \in \{2, \ldots, M\}$ to party P_i. The insecure implementation of this functionality is obtained by sending the PRF key to each party to locally evaluate the PRF at its own index location. For $\epsilon \in (0, 1)$ and $S = M^\epsilon$, this can be implemented in constant number of rounds by distributing the PRF key in a (arity $\sqrt{S}$) tree-like manner. This protocol is clearly insecure (w.r.t any reasonable notion of security). We are going to show that in any semi-malicious implementation of this functionality, party P_1 must send $\Omega(M)$ bits of information throughout the execution.

The formal proof of the above intuition shows that any generic semi-malicious compiler must incur non-trivial overhead either in space or in the round complexity (thereby making our protocol not in the MPC model). This is formalized in the full version of our paper, and Theorem 4.1 is a direct corollary of it. The proof is an adaptation of [44] and is given for completeness.

We refer to the full version for the proof of Theorem 4.1.

5 Common Subprotocols

We introduce two common subprotocols that take $O(\log_\gamma M)$ rounds and the communication is $O(S \cdot \gamma)$ per round for each machine, implement useful functionalities.

5.1 The Distribute Subprotocol

Consider the simple distribution functionality: P_1 has some string x and it wants to distribute x to all the other parties. In the normal model with point-to-point channels, P_1 can just send x to every other party which can be done in a single round. However, is problematic since it requires P_1 to send messages of $\Omega(M)$ bits in a single round. The following protocol implements this functionality by delivering x along a "tree".

Protocol 1 $\mathsf{Distribute}_\gamma(x)$

Input: P_1 holds a string x where $|x| \leq S$.
Output: Each party holds x.

1: Let $t = \lceil \log_\gamma M \rceil$. We refer to a party P_i as being on the level k if $(i-1)$ is a multiple of γ^k.
2: For each round $k \in [t]$, all the parties on level $t + 1 - k$ send x to the parties on level $t - k$.

5.2 The Combine Subprotocol

The protocol Combine (described in Protocol 2) implements the following functionality. Initially, each party $i \in [M]$ has an input x_i, and they want to jointly compute $\mathsf{op}_{i=1}^{M} x_i = x_1 \ \mathsf{op} \ x_2 \ \mathsf{op} \ \ldots \ \mathsf{op} \ x_M$, where op is some associative operator. Note that if each party i sends x_i to the recipient P_1 in a single round, P_1 receive messages of $\Omega(M)$ bits in a single round. In protocol Combine, we use the similar

trick to ask parties aggregate the values in a tree fashion and in each round, all child nodes send the values they aggregate in their own subtree to their parent nodes.

Protocol 2 $\mathsf{Combine}_\gamma(\mathsf{op}, \{x_i\}_{i\in[M]})$

Input: Party P_i holds x_i where $\gamma \cdot |x_i| \leq S$, and the parties agree on an associative operator op.

Output: P_1 holds $\mathsf{op}_{i=1}^{M} x_i$.

1: Let $t = \lceil \log_\gamma M \rceil$. We refer to a party P_i as being on the level k if $(i-1)$ is a multiple of γ^k. Each node P_i sets $x_{i,0} \leftarrow x_i$.
2: For each round $k \in [t]$, for each party i on level k, P_i computes $x_{i,k} = \mathsf{op}_{j=1}^{\gamma} x_{j',k-1}$ where $j' = i + \gamma^{k-1}(j-1)$.
3: After t rounds, P_1 has $x_{1,t} = x_1 \,\mathsf{op}\, \ldots \,\mathsf{op}\, x_M$.

6 Semi-Malicious Secure MPC for Long Output

In this section, we give a semi-malicious compiler for general MPC protocols. The compiler takes as input an arbitrary (possibly insecure) MPC protocol and transforms it into a semi-malicious counterpart.

Theorem 6.1 (Semi-Malicious Secure MPC for Long Output). *Let $\lambda \in \mathbb{N}$ be a security parameter. Assume that we are given a deterministic MPC protocol Π that completes in R rounds in which each of the M machines utilizes at most S local space. Assume that $M \in \mathsf{poly}(\lambda)$ and $\lambda \leq S$. Further, assume that there is a (non-leveled) threshold FHE scheme .*

Then, there is a compiler that transforms Π into another protocol $\tilde{\Pi}$ which assumes a PKI and a (programmable) random oracle, and furthermore realizes Π with P2P semi-malicious security *in the presence of an adversary that statically corrupts up to $M-1$ parties. Moreover, $\tilde{\Pi}$ completes in $R + O(1)$ rounds and consumes at most $S \cdot \mathsf{poly}(\lambda)$ space per machine.*

Property of our compiler is that for every message m, that sent in the original protocol, the size of the corresponding message in compiled protocol is $|m| \cdot \mathsf{poly}(\lambda)$, and the size of every additional message in the compiled protocol is $\mathsf{poly}(\lambda)$.

Our compiler also support different space per machine. Specifically, let S_i be the space of the corresponding machine in the original protocol, then this machine consumes at most $S_i \cdot \mathsf{poly}(\lambda)$ space in the compiled protocol. Similarly, the communication complexity of each machine is also preserved, up to the same multiplicative security parameter blowup.

We refer to the full version for the proof of Theorem 6.1.

7 Malicious-Secure MPC

This section is devoted to presenting and analyzing our P2P-semi-malicious to malicious compiler. The formal statement is given next.

Theorem 7.1 (P2P-Semi-malicious to malicious compiler). *Assume hardness of LWE and the existence of a SNARK scheme for NP. Let $\lambda \in \mathbb{N}$ be a security parameter. Assume that we are given a P2P-semi-malicious MPC protocol Π secure against up to $M-1$ corruptions in the PKI model. Suppose that it consumes R rounds in which each of the M machines utilizes at most S local space. Assume that $M \in \mathsf{poly}(\lambda)$ and $\lambda \leq S$.*

Then, there exists an MPC protocol which is maliciously secure against up to $M-1$ corruptions in the PKI model which realizes the same functionality. Moreover, the compiled protocol completes in $O(R)$ rounds and consumes at most $S \cdot \mathsf{poly}(\lambda)$ space per party.

Combining Theorem 7.1 together with our short output semi-malicious compiler (which is given in the full version of the paper), we obtain a maliciously secure compiler for short output deterministic MPC protocols. Combining Theorem 7.1 together with our long output semi-malicious compiler from Theorem 6.1 we obtain a maliciously secure compiler for arbitrary deterministic MPC protocols. Full details are given in Section 7.3.

The rest of the section is organized as follows. In Section 7.1, we define several subprotocols which we will use. In Section 7.2, we give the formal description of the compiler and analyze its efficiency. Finally, in Section 7.3 we describe how to put all the pieces together to obtain a long-output malicious-secure compiler. We defer a formal proof of security to the full version of our paper.

7.1 The Subprotocols

We mention the subprotocols which will be used. These subprotocols enable the parties to compute and agree upon a Merkle root which commits to a concatenation of all parties' inputs, and to compute a succinct proof of honest behavior for each round of the underlying protocol. Note that the compiler and its subprotocols both use the Distribute and Combine subprotocols defined in Section 5. Due to lack of space, we briefly explain the subprotocols here and refer to the full version for full details.

The CalcMerkleTree *Subprotocol.* The purpose of this protocol is for all parties to know a Merkle root τ with respect to some hash function h which commits to their collective inputs, and for each party P_i to know an opening θ_i for its respective input. We will perform this process over a tree with arity γ. The process completes within $2\lceil \log_\gamma M \rceil$ rounds, where in each round the current layer calculates new labels and sends them to the new layer of parents, and each layer sends any opening $\theta_{i,j}$ received from its parent to all its children. At the end, each party P_i will know the root τ and an opening π_i to x_i.

The Agree *Subprotocol.* When using the CalcMerkleTree subprotocol in the malicious setting, it is not guaranteed that all honest parties will receive a consistent Merkle root τ. Indeed, the corrupted parties could cause different honest parties to receive different roots, or could prevent some honest parties from learning

the openings for their inputs. Because of this, we need a way for all parties to agree on a single root, and for parties to be able to force an abort if they did not receive valid openings. To that end, we define the subprotocol Agree. In this subprotocol, each party P_i has as input a string x_i. The subprotocol aborts if there exists i, j where $x_i \neq x_j$. The main primitive used is a threshold signature scheme with distributed reconstruction (TSDR), see the full version of the paper for a formal definition. The distributed reconstruction property is used to achieve the required space efficiency properties.

The SNARK statements and the RecCompAndVerify *subprotocol.* The last subprotocol, RecCompAndVerify, deals with recursive composition and verification of the zkSNARKs that prove honest behavior during the commitment phase and during each round of the underlying protocol. The subprotocol recursively composes proofs of honest behavior of each party in a given round to get a succinct joint proof of all parties' honest behavior in that round. The parties then verify the proof and abort if the proof fails to verify.

The statements used when computing zkSNARKs is $\Phi((i, r, 0, \tau_{r-1}, \tau_r), w)$. It proves that P_i's state in τ_r was computed honestly with respect to its state in τ_{r-1}, and that it sent honest messages to every party it was supposed to send messages to during round r. The security properties needed for RecCompAndVerify are defined via a game RCVSecurity. In this game, a nonuniform PPT adversary $\mathcal{A}$ invokes R sequential instances of RecCompAndVerify. The game takes two parameters r_1 and r_2; the challenger will try to extract from the proofs produced by $\mathcal{A}$ during the r_1-th and r_2-th RecCompAndVerify instances. The game is defined this way to support the extraction requirements during the proof of security of the main compiler, which is designed to only need to extract twice during the protocol.

7.2 The Compiler

We now give the formal description of the compiler.

Protocol 3 Malicious-Secure Compiler

Setup: Each party P_i knows the verification key vk along with secret key ssk_i, where $(\mathsf{vk}, \mathsf{ssk}_1, \ldots, \mathsf{ssk}_m) \leftarrow \mathsf{Sig.Setup}(1^\lambda, 1^M)$ are the setup parameters for the TSDR scheme. The parties also know a hash function h and a SNARK CRS crs. Finally, the parties know the P2P semi-malicious setup: every party knows the semi-malicious public key smpk, and each party P_i knows its semi-malicious secret key smsk_i.

Input: Party P_i has input x_i and randomness r_i to the underlying MPC protocol.

Commitment Phase:

1. Each party P_i chooses a PRF key k_i and computes a commitment $c_{k_i} \leftarrow \mathsf{C.Commit}(k_i; \alpha_{k_i})$. It then computes $c_{\mathsf{st}_{i,0}} \leftarrow \mathsf{C.Commit}((x_i, r_i); \mathsf{PRF}_{k_i}(0))$.
2. The parties run the subprotocol $\mathsf{CalcMerkleTree}_h(\{c_{\mathsf{st}_{i,0}} || c_{k_i}\}_{i\in[n]})$, so that each party P_i obtains a Merkle commitment τ_0 and an opening $\theta_{i,0}$ to $c_{\mathsf{st}_{i,0}} || c_{k_i}$. Party P_i aborts if its opening is not valid.
3. The parties run the subprotocol $\mathsf{Agree}_\lambda((0, \tau_0), \mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]})$ and abort if the subprotocol aborts.
4. Each party P_i calculates a SNARK $\pi_{0,r} \leftarrow \Pi.\mathsf{P}(crs, \Phi(i, 0, \bot, \tau_0), (\bot, \bot, \bot, c_{\mathsf{st}_{i,0}}, \bot, \bot, \theta_{i,0}, k_i, c_{k_i}, \alpha_{k_i}, \bot, \mathsf{st}_{i,0}, \bot), (i, 0))$.
5. All parties run $\mathsf{RecCompAndVerify}(crs, \bot, \tau_0, \{\pi_{i,0}\}_{i\in[M]})$ to obtain and verify π_0, a SNARK for the statement $\Phi(0, 0, t, \bot, \tau_0)$. If the subprotocol aborts then all parties abort and stop responding.

Evaluation Phase: The evaluation phase is divided into steps corresponding to the rounds of the original protocol. Each step consists of several rounds in the new protocol. For each of the R steps, the behavior of each party P_i is as follows:

- **For round r of the underlying protocol:** P_i starts with a state $(\mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1}, \mathsf{msg}^{\mathsf{out}}_{i,r-1})$, a Merkle root τ_{r-1} for the previous round's global state, and an opening $\theta_{i,r-1}$ for $c_{\mathsf{st}_{i,r-1}} || \mathsf{msg}^{\mathsf{in}}_{i,r-1} || \mathsf{msg}^{\mathsf{out}}_{i,r-1} || c_{k_i}$ with respect to τ_{r-1}, where $c_{\mathsf{st}_{i,r-1}} = \mathsf{C.Commit}(\mathsf{st}_{i,r-1}; \mathsf{PRF}_{k_i}(r-1))$.
  1. Compute $(\mathsf{st}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r}) \leftarrow \mathsf{NextSt}_{i,r}(\mathsf{smpk}, \mathsf{smsk}_i, \mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$.
  2. For each $(j, s_j, e_j) \in \mathsf{OutgoingMessageLocs}(i, r)$, send $\mathsf{msg}^{\mathsf{out}}_{i,r}[s_j : e_j]$ to party P_j.
  3. Initialize $\mathsf{msg}^{\mathsf{in}}_{i,r}$ as an empty string of the appropriate size.
  4. For each message m received from party j during the last step, write m to $\mathsf{msg}^{\mathsf{in}}_{i,r}$ at location $\mathsf{IncomingMessageLoc}(j, i, r)$.
  5. Compute $c_{\mathsf{st}_{i,r}} \leftarrow \mathsf{C.Commit}(\mathsf{st}_{i,r}; \mathsf{PRF}_{k_i}(r))$.
  6. Run $\mathsf{CalcMerkleTree}_h(\{c_{\mathsf{st}_{i,r}} || \mathsf{msg}^{\mathsf{in}}_{i,r} || \mathsf{msg}^{\mathsf{out}}_{i,r} || c_{k_i}\}_{i\in[M]})$ with all other parties to obtain τ_r, the Merkle root of the transcript, along with $\theta_{i,r}$, an opening to $\mathsf{st}_{i,r} || \mathsf{msg}^{\mathsf{in}}_{i,r} || \mathsf{msg}^{\mathsf{out}}_{i,r} || c_{k_i}$ with respect to τ_r. Abort if the opening is not valid.
  7. Run $\mathsf{Agree}_\lambda((r, \tau_r), \mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]})$ and abort if the subprotocol aborts.
  8. For each party that sent a message to P_i, send $\theta_{m_{i,r}}$, an opening to position $\mathsf{IncomingMessageGlobalLoc}(j, i, r)$ in τ_r.
  9. Calculate a SNARK $\pi_{r,i} \leftarrow \Pi.\mathsf{P}(crs, \Phi((i, r, 0, \tau_{r-1}, \tau_r), (c_{\mathsf{st}_{i,r-1}}, \mathsf{msg}^{\mathsf{in}}_{i,r-1}, \mathsf{msg}^{\mathsf{out}}_{i,r-1}, \theta_{i,r-1}, c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{in}}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r}, \theta_{i,r}, k_i, c_{k_i}, \alpha_{k_i}, \mathsf{st}_{i,r-1}, \mathsf{st}_{i,r}, \{(m_{j,r}, \theta_{m_{j,r}})\}_j)), (i, 0))$.
  10. All parties run $\mathsf{RecCompAndVerify}(crs, \tau_{r-1}, \tau_r, \{\pi_{i,r}\}_{i\in[M]})$ to obtain π_r, a SNARK for statement $\Phi(0, r, t, \tau_{r-1}, \tau_r)$. If the subprotocol aborts, then all parties abort and stop responding.

Output Phase: At the end of round R, each player P_i has a state $(\mathsf{st}_{i,R}, \mathsf{msg}^{\mathsf{in}}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r})$. P_i does the following to compute its final output:

1. Compute $y_i \leftarrow \mathsf{NextSt}_{i,R}(\mathsf{smpk}, \mathsf{smsk}_i, \mathsf{st}_{i,R}, \mathsf{msg}^{\mathsf{in}}_{i,r})$.
2. Output y_i.

Correctness and efficiency. Correctness of the compiler follows directly from the correctness of the underlying building blocks. To analyze the efficiency of the compiler, we first recall that during the sub-protocols $\mathsf{CalcMerkleTree}$, Agree, and $\mathsf{RecCompAndVerify}$, each machine takes local space bounded by $S \cdot \mathsf{poly}(\lambda)$. Moreover, the complexity-preserving efficiency property of the idse-zkSNARK scheme guarantees that $\Pi.\mathsf{P}(crs, \phi, w)$ is proportional to $\mathsf{poly}(\lambda) \cdot (|\phi|+|w|+s)$, where s is the maximum space of the verification procedure for ϕ. Finally, when carrying over between rounds, the parties only need to remember the previous round's Merkle root and an opening of size $\mathsf{poly}(\lambda) \cdot S$ along with k_i and the randomness used to generate the commitment c_{k_i}. It follows from these three facts that if the total local space used by each machine during the original protocol Π is S, then the total local space used by each machine during the compiled protocol $\tilde{\Pi}$ is at most $S \cdot \mathsf{poly}(\lambda)$.

7.3 Putting it All Together

Given a *short output* MPC protocol, we can directly compile it into a P2P semi-malicious secure protocol with our short output "insecure to P2P semi-malicious secure" compiler. Then, we can compile it into a maliciously secure protocol with our "P2P semi-malicious to malicious secure" compiler from Section 7. The resulting maliciously secure protocol has only constant overhead in round complexity and a $\mathsf{poly}(\lambda)$ blowup in space. This lead to the following corollary:

Corollary 1. *Assume the existence of a (non-leveled) threshold FHE system, LWE, and a SNARK scheme for NP. Let $\lambda \in \mathbb{N}$ be a security parameter. Assume that we are given a (insecure) deterministic* short output *MPC protocol Π. Suppose that it consumes R rounds in which each of the M machines utilizes at most S local space. Assume that $M \in \mathsf{poly}(\lambda)$ and $\lambda \leq S$.*

Then, there exists an MPC protocol which realizes the same functionality as Π and which is malicious secure against up to $M-1$ corruptions in the PKI model. Moreover, the compiled protocol completes in $O(R)$ rounds and consumes at most $S \cdot \mathsf{poly}(\lambda)$ space per party.

Given any long output protocol, we can compile it into a P2P semi-malicious secure protocol with our long output "insecure to P2P semi-malicious secure" compiler from Section 6. This results with a protocol in the random oracle model (which is somewhat inherent due to our lower bound). Unfortunately, we cannot directly use our "P2P semi-malicious to malicious secure" compiler since in the description of the latter we did not capture input protocols that rely on a random oracle. The reason is that SNARKs do not compose well with random oracles.

More specifically, in the long output compiled protocol all the parties calculate a shared string denoted r_{seed} , then each party calculates offline the root of a Merkle tree of the values $\{\mathcal{O}(r_{seed}||i)\}_{i\in[M]}$ which we denoted z_r . Our goal is to prove that z_r is correctly calculated.

Note that z_r is a deterministic function of r_{seed} (since the random oracle is deterministic during the execution of the protocol). So, z_r can be calculated offline and its size is $\mathsf{poly}(\lambda)$. Now, in the "P2P semi-malicious to malicious secure" compiler, after round r that corresponds to the end of in the long output compiled protocol, we perform the following steps:

1. (Recall that τ_r is the Merkle tree root of states and messages of all parties at round r.) In addition to storing τ_r, we also store z_r. Denote $\tau_r^* = (\tau_r, z_r)$ and from now on, use τ_r^* instead of τ_r.
2. The parties run $\mathsf{Agree}_\lambda(\tau_r^*, \mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]})$ and abort if the sub-protocol aborts.

The above steps guarantee that all of the parties use the same z_r. In round $r+1$ of the malicious compiled protocol, whenever a SNARK is computed , it proved that if we know that τ_r^* is correctly calculated, then it must also be the case that τ_{r+1} is correctly calculated. In particular, the SNARK is never applied on a statement that contains a random oracle query.

A different way to interpret the above is to imagine the statement provided to the SNARK as composed of two parts: one that depends on a short seed r_{seed} (that all parties know) and consists of random oracle queries which eventually result with a small digest z_r, and the other is a plain model computation that only depends on z_r. The point is that since z_r is deterministic function of r_{seed}, the random-oracle dependent calculation can be *locally* computed by each party (and so z_r can be verified) and the SNARK can be applied only to the plain model computation that depends on z_r. Overall, we obtain the following corollary.

Corollary 2. *Assume the existence of a (non-leveled) threshold FHE system, LWE, a SNARK scheme for NP, and iO. Let $\lambda \in \mathbb{N}$ be a security parameter. Assume that we are given a (insecure) deterministic MPC protocol Π. Suppose that it consumes R rounds in which each of the M machines utilizes at most S local space. Assume that $M \in \mathsf{poly}(\lambda)$ and $\lambda \leq S$.*

Then, there exists an MPC protocol which realizes the same functionality as Π and which is malicious secure against up to $M-1$ corruptions, in the PKI/RO model. Moreover, the compiled protocol completes in $O(R)$ rounds and consumes at most $S \cdot \mathsf{poly}(\lambda)$ space per party.

Acknowledgements. Rex Fernando is supported in part from a Simons Investigator Award, DARPA SIEVE award, NTT Research, NSF Frontier Award 1413955, BSF grant 2018393, a Xerox Faculty Research Award, a Google Faculty Research Award, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through Award HR00112020024. Yuval Gelles and Ilan Komargodski are supported in part by an Alon Young Faculty Fellowship, by a JPM Faculty Research Award,

by a grant from the Israel Science Foundation (ISF Grant No. 1774/20), and by a grant from the US-Israel Binational Science Foundation and the US National Science Foundation (BSF-NSF Grant No. 2020643). Elaine Shi is supported in part by the US National Science Foundation (NSF awards 2044679 and 2128519).

References

1. Ahn, K.J., Guha, S.: Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. TOPC (2018)
2. Andoni, A., Nikolov, A., Onak, K., Yaroslavtsev, G.: Parallel algorithms for geometric graph problems. In: STOC (2014)
3. Andoni, A., Stein, C., Zhong, P.: Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In: ICALP (2019)
4. Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. In: EUROCRYPT. pp. 483–501 (2012)
5. Assadi, S.: Simple round compression for parallel vertex cover. CoRR **abs/1709.04599** (2017)
6. Assadi, S., Bateni, M., Bernstein, A., Mirrokni, V., Stein, C.: Coresets meet edcs: algorithms for matching and vertex cover on massive graphs. arXiv preprint arXiv:1711.03076 (2017)
7. Assadi, S., Khanna, S.: Randomized composable coresets for matching and vertex cover. In: SPAA (2017)
8. Assadi, S., Sun, X., Weinstein, O.: Massively parallel algorithms for finding well-connected components in sparse graphs. CoRR **abs/1805.02974** (2018)
9. Badrinarayanan, S., Jain, A., Manohar, N., Sahai, A.: Threshold multi-key fhe and applications to round-optimal mpc. IACR Cryptology ePrint Archive p. 580 (2018)
10. Bahmani, B., Kumar, R., Vassilvitskii, S.: Densest subgraph in streaming and mapreduce. Proceedings of the VLDB Endowment **5**(5), 454–465 (2012)
11. Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S.: Scalable k-means++. Proceedings of the VLDB Endowment **5**(7), 622–633 (2012)
12. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM (2012)
13. Bateni, M., Bhaskara, A., Lattanzi, S., Mirrokni, V.: Distributed balanced clustering via mapping coresets. In: in NeurIPS (2014)
14. Behnezhad, S., Brandt, S., Derakhshan, M., Fischer, M., Hajiaghayi, M., Karp, R.M., Uitto, J.: Massively parallel computation of matching and MIS in sparse graphs. In: PODC (2019)
15. Behnezhad, S., Hajiaghayi, M., Harris, D.G.: Exponentially faster massively parallel maximal matching. In: FOCS (2019)
16. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC (1988)
17. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Algorithmica **79**(4), 1102–1160 (2017)
18. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKS and proof-carrying data. In: STOC (2013)
19. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: CRYPTO. pp. 565–596 (2018)

20. Boyle, E., Chung, K., Pass, R.: Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In: CRYPTO (2015)
21. Boyle, E., Jain, A., Prabhakaran, M., Yu, C.: The bottleneck complexity of secure multiparty computation. In: ICALP (2018)
22. Brakerski, Z., Perlman, R.: Lattice-based fully dynamic multi-key FHE with short ciphertexts. In: Advances in Cryptology - CRYPTO. pp. 190–213 (2016)
23. Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: Advances in Cryptology - CRYPTO. pp. 681–710 (2021)
24. Bünz, B., Chiesa, A., Mishra, P., Spooner, N.: Recursive proof composition from accumulation schemes. In: Theory of Cryptography - TCC. pp. 1–18 (2020)
25. Chan, T.H., Chung, K., Lin, W., Shi, E.: MPC for MPC: secure computation on a massively parallel computing architecture. In: ITCS (2020)
26. Chang, Y., Fischer, M., Ghaffari, M., Uitto, J., Zheng, Y.: The complexity of (Δ+1) coloring in congested clique, massively parallel computation, and centralized local computation. In: PODC (2019)
27. Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: Innovations in Computer Science - ICS. pp. 310–331 (2010)
28. Chiesa, A., Tromer, E., Virza, M.: Cluster computing in zero knowledge. In: Advances in Cryptology - EUROCRYPT. pp. 371–403 (2015)
29. Czumaj, A., Łącki, J., Mądry, A., Mitrović, S., Onak, K., Sankowski, P.: Round compression for parallel matching algorithms. In: STOC (2018)
30. Dodis, Y., Halevi, S., Rothblum, R.D., Wichs, D.: Spooky encryption and its applications. In: Robshaw, M., Katz, J. (eds.) CRYPTO (2016)
31. Ene, A., Im, S., Moseley, B.: Fast clustering using mapreduce. In: SIGKDD (2011)
32. Ene, A., Nguyen, H.: Random coordinate descent methods for minimizing decomposable submodular functions. In: ICML (2015)
33. Fernando, R., Komargodski, I., Liu, Y., Shi, E.: Secure massively parallel computation for dishonest majority (2020)
34. Gamlath, B., Kale, S., Mitrovic, S., Svensson, O.: Weighted matchings via unweighted augmentations. In: PODC (2019)
35. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS (2013)
36. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM Symposium on Theory of Computing, STOC. pp. 169–178 (2009)
37. Gentry, C., Wichs, D.: Separating succinct non-interactive arguments from all falsifiable assumptions. In: Fortnow, L., Vadhan, S.P. (eds.) STOC (2011)
38. Ghaffari, M., Lattanzi, S., Mitrović, S.: Improved parallel algorithms for density-based network clustering. In: ICML (2019)
39. Ghaffari, M., Uitto, J.: Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In: SODA (2019)
40. Goldreich, O.: Foundations of cryptography: volume 2. Cambridge university press (2009)
41. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC (1987)
42. Groth, J.: On the size of pairing-based non-interactive arguments. In: EUROCRYPT (2016)
43. Hajiaghayi, M., Seddighin, S., Sun, X.: Massively parallel approximation algorithms for edit distance and longest common subsequence. In: SODA (2019)
44. Hubáček, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: ITCS. pp. 163–172 (2015)

45. Karloff, H.J., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: SODA (2010)
46. Katz, J., Ostrovsky, R., Smith, A.D.: Round efficiency of multi-party computation with a dishonest majority. In: Eurocrypt (2003)
47. Kumar, R., Moseley, B., Vassilvitskii, S., Vattani, A.: Fast greedy algorithms in mapreduce and streaming. TOPC **2**(3), 14:1–14:22 (2015)
48. Łącki, J., Mirrokni, V.S., Wlodarczyk, M.: Connected components at scale via local contractions. CoRR **abs/1807.10727** (2018)
49. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce. In: SPAA (2011)
50. Lindell, Y., Nissim, K., Orlandi, C.: Hiding the input-size in secure two-party computation. In: Advances in Cryptology - ASIACRYPT. pp. 421–440 (2013)
51. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
52. Micali, S.: CS proofs (extended abstracts). In: FOCS (1994)
53. Mirzasoleiman, B., Karbasi, A., Sarkar, R., Krause, A.: Distributed submodular maximization: Identifying representative elements in massive data. In: NeurIPS (2013)
54. Mukherjee, P., Wichs, D.: Two round multiparty computation via multi-key FHE. In: Fischlin, M., Coron, J. (eds.) Eurocrypt (2016)
55. Naor, M., Nissim, K.: Communication preserving protocols for secure function evaluation. In: STOC. pp. 590–599 (2001)
56. Onak, K.: Round compression for parallel graph algorithms in strongly sublinear space. CoRR **abs/1807.08745** (2018)
57. Pass, R.: Bounded-concurrent secure multi-party computation with a dishonest majority. In: Babai, L. (ed.) STOC (2004)
58. Peikert, C., Shiehian, S.: Multi-key FHE from LWE, revisited. In: TCC (2016)
59. da Ponte Barbosa, R., Ene, A., Nguyen, H.L., Ward, J.: A new framework for distributed submodular maximization. In: FOCS. pp. 645–654 (2016)
60. Rastogi, V., Machanavajjhala, A., Chitnis, L., Sarma, A.D.: Finding connected components in map-reduce in logarithmic rounds. In: ICDE (2013)
61. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009)
62. Yaroslavtsev, G., Vadapalli, A.: Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. In: ICML (2018)