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`.

August 8, 2022

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.

Contents

1 Introduction

The Massively Parallel Computation (MPC[1]) model, first introduced by Karloff, Suri, and Vassilvitskii [KSV10], 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 [AG18, ANOY14, ASZ19, Ass17, ABB$^+$17, AK17, ASW18, BKV12, BHH19, BBD$^+$19, CFG$^+$19, CŁM$^+$18, GU19, GKMS19, ŁMW18, LMSV11, Ona18, RMCS13, HSS19], clustering [BMV$^+$12, BBLM14, EIM11, GLM19, YV18] and submodular function optimization [dPBENW16, EN15, KMVV15, MKSK13].

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* 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

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

multiparty computation (starting with [GMW87, BGW88]), 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 [GMW87, KOS03, Pas04, AJL$^+$12, MW16, DHRW16]. Also, protocols that require simultaneously sending one broadcast message per party (e.g., Boyle et al. [BCP15]) are unfit since this also implies that each party needs to receive and store a message from all other parties.[2]

State of the art. Chan, Chung, Lin, and Shi [CCLS20] 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 [Reg09, AJL$^+$12, BGG$^+$18]. Otherwise, we need a non-leveled scheme which we can get by relying on Gentry's bootstrapping technique [Gen09]. This requires the TFHE scheme being "circular secure".[3]

More recently, Fernando et al. [FKLS20] 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 [BGI$^+$12, GGH$^+$13]. 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?

[2]Some works design "communication preserving" secure computation protocols (for example, [NN01, LNO13, HW15]) where the goal is to eliminate input/output-size dependency in communication complexity—all of these works only address the two parties setting.

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

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. [FKLS20], 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 [FKLS20]'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., [ASZ19, ASW18, GU19, HSS19] 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. [FKLS20], 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 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 [HW15] 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 constructions of SNARKs with various trade offs between efficiency, security guarantees, and the required assumptions, for example the works of [Mic94, BCCT13, Gro16]. 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. [FKLS20] 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. [FKLS20]. Indeed, we get the same result as that of [FKLS20] 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. [AJL$^+$12], 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 [GMW87] 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., [AJL$^+$12, MW16]) 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. Section 3 contains preliminaries for the rest of the paper. In Section 4 we formally defined the model and the malicious and P2P-semi-malicious security definitions. In Section 5 we prove the impossibility result of generic (semi-)malicious compilers even using setup assumptions. In Section 6 we introduce two commonly used procedures. In Section 7 we give a P2P-semi-malicious compiler for long-output MPC protocols. Lastly, in Section 8 we give our P2P-semi-malicious-to-malicious compiler. For completeness, in Appendix A, we give a P2P-semi-malicious compiler for short-output MPC protocols. The protocol is essentially the same as [FKLS20], however we give an updated proof to achieve the stronger security definition.

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 $f\colon (\{0,1\}^{l_{\mathsf{in}}})^M \to (\{0,1\}^{l_{\mathsf{out}}})^M$, 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. [CCLS20] 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. [FKLS20] 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) [GMW87]. 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 [HW15] 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. [FKLS20], 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. [FKLS20]: 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. [FKLS20] 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 [AJL$^+$12, LTV12, MW16, BP16, PS16, BGG$^+$18, BJMS18] 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.*

2.1.1 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 [AJL$^+$12, MW16], 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.

2.1.2 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 [GMW87] 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 [BCCT13].*

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, [BCCT13, CTV15, BCTV17, BJPY18]). We want to avoid such strong assumptions.

Remark 2. *At a very high level, proofs carrying data (PCDs) [CT10, BCCT13, BCMS20, BCL$^+$21] 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. [BCCT13] 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. $\mathsf{CalcMerkleTree}$ 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. [BGG$^+$18] to our setting. Crucially, the signature scheme of [BGG$^+$18] 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. $\mathsf{RecCompAndVerify}$ 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 $\mathsf{CalcMerkleTree}$ 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 [BJPY18]. 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. [BJPY18]. (Note that the transformation of [BCCT13] 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 [GW11]. 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 8.

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. [FKLS20].

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. [FKLS20] 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 [HW15] 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.[4]

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 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 7 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 2.1.2. 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

[4]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.

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 Preliminaries

For $x \in \{0,1\}^*$, let $x[a : b]$ be the substring of x starting at a and ending at b. A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}$ is *negligible* if it is asymptotically smaller than any inverse-polynomial function, namely, for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) \leq \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for any non-uniform PPT algorithm $\mathcal{A}$ there exists a negligible function negl such that $\left|\Pr[\mathcal{A}(1^\lambda, X_\lambda) = 1] - \Pr[\mathcal{A}(1^\lambda, Y_\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$.

3.1 Somewhere Statistically Binding Hash

A somewhere statistically binding (SSB) hash [HW15] consists of the following algorithms, which satisfy the properties below:

- $\mathsf{SSB.Setup}(1^\lambda, L, d, f, i^*) \to h$: On input integers L, d, f, and an index $i^* \in [f^d L]$, outputs a hash key h.
- $\mathsf{SSB.Start}(h, x) \to v$: On input h and a string $x \in \{0,1\}^L$, output a hash tree leaf v.
- $\mathsf{SSB.Combine}(h, \{v_i\}_{i\in[f]}) \to \hat{v}$: On input h and f hash tree nodes $\{v_i\}_{i\in[f]}$, output a parent node $\hat{v}$.
- $\mathsf{SSB.Verify}(h, i, x_i, z, \{v\}) \to b$: On input h, and index i, a string x_i, a hash tree root z, and a set $\{v\}$ of nodes, output 1 iff $\{v\}$ consists of a path from the leaf corresponding to x_i to the root z, as well as the siblings of all nodes along this path.

Correctness: For any integers L, d, and f, and any indices i^*, j, strings $\{x_i\}_{i\in[f^d]}$ where $|x_i| = L$, and any $h \leftarrow \mathsf{SSB.Setup}(1^\lambda, L, d, f, i^*)$, if $\{v\}$ consists of a path in the tree generated using $\mathsf{SSB.Start}(h, \cdot)$ and $\mathsf{SSB.Combine}(h, \cdot)$ on the leaf strings $\{x_i\}_{i\in[f^d]}$, from the leaf corresponding to x_j to the root z, along with the siblings of all nodes along this path, then $\mathsf{SSB.Verify}(h, j, x_j, z, \{v\}) = 1$.

Compactness of commitment and openings: All node labels generated by the $\mathsf{SSB.Start}$ and $\mathsf{SSB.Combine}$ algorithms are binary strings of size $\mathsf{poly}(\lambda) \cdot L$.

Index hiding: Consider the following game between an adversary $\mathcal{A}$ and a challenger:

1. $\mathcal{A}(1^\lambda)$ chooses L, d, and f, and two indices i_0^* and i_1^*.
2. The challenger chooses a bit $b \leftarrow_\$ \{0,1\}$ and sets $h \leftarrow \mathsf{SSB.Setup}(1^\lambda, L, d, f, i_b^*)$.
3. The adversary gets h and outpus a bit b'. The game outputs 1 iff $b = b'$.

We require that no PPT $\mathcal{A}$ can win the game with non-negligible probability.

Somewhere statistically binding: For all λ, L, d, and f, i^*, and for any key $h \leftarrow \mathsf{SSB.Setup}(1^\lambda, L, d, f, i^*)$, there do not exist any values z, x, x', $\{v\}$, $\{v'\}$ such that $\mathsf{SSB.Verify}(h, i^*, x, z, \{v\}) = \mathsf{SSB.Verify}(h, i^*, x', z, \{v'\}) = 1$.

Theorem 3.1 ([HW15, Theorem 3.2])**.** *Assume LWE. Then there exists an SSB hash construction satisfying the above properties.*

3.2 Indistinguishability Obfuscation for Circuits

Let $\mathcal{C}$ be a class of Boolean circuits. An obfuscation scheme for $\mathcal{C}$ consists of one algorithm iO with the following syntax.

$iO(C \in \mathcal{C}, 1^\lambda)$: The obfuscation algorithm is a PPT algorithm that takes as input a circuit $C \in \mathcal{C}$, security parameter λ. It outputs an obfuscated circuit.

An obfuscation scheme is said to be a secure indistingushability obfuscator for $\mathcal{C}$ [BGI+12, GGH+13, SW14] if it satisfies the following correctness and security properties:

- Correctness: For every security parameter λ, input length n, circuit $C \in \mathcal{C}$ that takes n bit inputs, input $x \in \{0,1\}^n$, $C'(x) = C(x)$, for $C' \leftarrow iO(C, 1^\lambda)$.
- Security: For every PPT adversary $\mathcal{A} = (A_1, A_2)$, the following experiment outputs 1 with at most $1/2 + \mathsf{negl}(\lambda)$:

  Protocol 1 (Experiment $\mathsf{Expt}_{\mathcal{A},\mathrm{iO}}$:)**.** *1.* $(C_0, C_1, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$

  2. If $|C_0| \neq |C_1|$*, or if either* C_0 *or* C_1 *have different input lengths, then the experiment outputs a uniformly random bit.*
  Else, let n *denote the input lengths of* C_0, C_1*. If there exists an input* $x \in \{0,1\}^n$ *such that* $C_0(x) \neq C_1(x)$*, then the experiment outputs a uniformly random bit.*

  3. $b \leftarrow \{0,1\}$*,* $\tilde{C} \leftarrow iO(C_b, 1^\lambda)$*.*

  4. $b' \leftarrow \mathcal{A}_2(\sigma, \tilde{C})$*.*

  5. Experiment outputs 1 *if* $b = b'$*, else it outputs* 0*.*

3.3 Puncturable Pseudorandom Functions

We use the definition of puncturable PRFs given in [SW14], given as follows. A puncturable family of PRFs F is given by a triple of turing machines $\mathsf{PPRF.KeyGen}$, $\mathsf{PPRF.Puncture}$, and F, and a pair of computable functions $n()$ and $m()$, satisfying the following conditions:

- **Functionality preserved under puncturing:** For every PPT adversary $\mathcal{A}$ such that $\mathcal{A}(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^{n(\lambda)}$, then for all $x \in \{0,1\}^{n(\lambda)}$ where $x \notin S$, we have that:

$$\Pr\left[F(K,x) = F(K_S,x) \mid \begin{array}{l} K \leftarrow \mathsf{PPRF.KeyGen}(1^\lambda), \\ K_S \leftarrow \mathsf{PPRF.Puncture}(K,S) \end{array}\right] = 1$$

- **Pseudorandom at punctured points:** For every PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ such that $\mathcal{A}_1(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^{n(\lambda)}$ and state σ, consider an experiment where $K \leftarrow \mathsf{PPRF.KeyGen}(1^\lambda)$ and $K_S \leftarrow \mathsf{PPRF.Puncture}(K,S)$. Then we have

$$\Big|\Pr\left[\mathcal{A}_2(\sigma, K_S, S, F(K,S)) = 1\right] - \Pr\left[\mathcal{A}_2(\sigma, K_S, S, U_{m(\lambda)|S|}) = 1\right]\Big| \leq \mathsf{negl}(\lambda)$$

where $F(K,S)$ denotes the concatenation of $F(K,x)$ for all $x \in S$ in lexicographic order and U_ℓ denotes the uniform distribution over ℓ bits.

Theorem 3.2 ([GGM84, BW13, BGI14, KPTZ13])**.** *If one-way functions exist, then for all efficiently computable $n(\lambda)$ and $m(\lambda)$ there exists a puncturable PRF family that maps $n(\lambda)$ bits to $m(\lambda)$ bits.*

3.4 M-out-of-M Threshold Fully Homomorphic Encryption

An M-out-of-M threshold fully homomorphic encryption scheme can be defined by the following polynomial-time algorithms:

- $\mathsf{TFHE.Setup}(1^\lambda, 1^d, M) \leftarrow (\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_M)$: On input the security parameter λ, the depth bound d, and the number of parties M, output a public key pk, a set of secret key shares, and the public parameters which we assume the other algorithms get as input implicitly. Furthermore, we assume sk is additively secret-shared such that $\mathsf{sk} = \sum_{i\in[M]} \mathsf{sk}_i$.
- $\mathsf{TFHE.Enc}(\mathsf{pk}, \mu) \rightarrow ct$: On input a public key pk and a message $\mu \in \{0,1\}^*$, output an encryption ct of the message.
- $\mathsf{TFHE.Eval}(C, \{ct_i\}_{i\in[k]}) \rightarrow ct_o$: On input a circuit $C\colon \{0,1\}^{l_1} \times \ldots \times \{0,1\}^{l_k} \rightarrow \{0,1\}^{l_o}$ of depth at most d and a set of ciphertexts $ct_1, \ldots, ct_k$, output a ciphertext ct_o.
- $\mathsf{TFHE.Dec}(\mathsf{sk}, ct) \rightarrow \mu$: On input a secret key sk and a ciphertext ct, output the decryption μ.
- $\mathsf{TFHE.PartDec}(\mathsf{sk}_i, ct) \rightarrow p_i$: On input a share of a secret key and a ciphertext ct, output a partial decryption p_i.
- $\mathsf{TFHE.Dec.Round}(p) \rightarrow \mu$: On input an (aggregated) partial decryption p, output a plaintext μ.

We require the following properties:

Definition 1 (Compactness)**.** *For all $\lambda, d, M \in \mathbb{N}$, all circuits $C\colon \{0,1\}^{l_1} \times \ldots \times \{0,1\}^{l_k} \rightarrow \{0,1\}^{l_o}$ of depth at most d, and all messages $\{\mu_i\}_{i\in[k]}$, the following holds. For any $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[M]}) \leftarrow \mathsf{TFHE.Setup}(1^\lambda, 1^d, M)$, $ct_j \leftarrow \mathsf{TFHE.Enc}(\mathsf{pk}, \mu_j)$ for $j \in [k]$, $ct_{\mathsf{out}} = \mathsf{TFHE.Eval}(C, \{ct_j\}_{j\in[k]})$, $p_j \leftarrow \mathsf{TFHE.PartDec}(\mathsf{sk}_j, ct_{\mathsf{out}})$, it holds that*

$$|\mathsf{pk}|, |\mathsf{sk}_i|, |ct_j|, |ct_{\mathsf{out}}|, |p_j| \leq \mathsf{poly}(\lambda, d, \log M)$$

for all $i \in [M], j \in [k]$.

Definition 2 (Correctness)**.** *For all $\lambda, d, M \in \mathbb{N}$, all circuits $C\colon \{0,1\}^{l_1} \times \ldots \times \{0,1\}^{l_k} \rightarrow \{0,1\}^{l_o}$ of depth at most d, and all messages $\{\mu_i\}_{i\in[k]}$, the following holds. For any $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[M]}) \leftarrow \mathsf{TFHE.Setup}(1^\lambda, 1^d, M)$, $\mathsf{sk} \leftarrow \sum_{i\in[M]} \mathsf{sk}_i$, $ct_j \leftarrow \mathsf{TFHE.Enc}(\mathsf{pk}, \mu_j)$ for $j \in [k]$, $ct_{\mathsf{out}} = \mathsf{TFHE.Eval}(C, \{ct_i\}_{i\in[k]})$, $p_i \leftarrow \mathsf{TFHE.PartDec}(\mathsf{sk}_i, ct_{\mathsf{out}})$, it holds that*

$$\Pr[\mathsf{TFHE.Dec}(\mathsf{sk}, ct_{\mathsf{out}}) = C(\mu_1, \mu_2, \ldots, \mu_k)] = 1 - \mathsf{negl}(\lambda),$$

and

$$\Pr[\mathsf{TFHE.Dec.Round}(\sum_{i=1}^{M} p_j) = C(\mu_1, \mu_2, \ldots, \mu_k)] = 1 - \mathsf{negl}(\lambda).$$

Definition 3 (Semantic Security)**.** *For all $\lambda, d, M \in \mathbb{N}$, the following holds. For any* PPT *adversary $\mathcal{A}$, the following experiment outputs 1 with negligible probability:*

Game 1 $\mathsf{Expt}_{\mathcal{A},\mathsf{TFHE},\mathsf{sem}}(1^\lambda, 1^d, M)$

```
1: On input the security parameter $\lambda$, the depth bound $d$, and the number of parties $M$, the
   adversary $\mathcal{A}$ output $\mathcal{C} \subsetneq [M]$ denoting the parties that have been corrupted.
2: The challenger runs $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[M]}) \leftarrow \mathsf{TFHE.Setup}(1^\lambda, 1^d, M)$.
3: The challenger samples $b \leftarrow \{0,1\}$ and provides $\mathsf{pk}$, $\mathsf{TFHE.Enc}(\mathsf{pk}, b)$, and $\{\mathsf{sk}_i\}_{i\in\mathcal{C}}$ to $\mathcal{A}$.
4: $\mathcal{A}$ outputs a guess $b'$. The experiment outputs 1 if $b = b'$.
```

We also require a TFHE scheme to satisfy *simulation security.* Informally, there exists a simulator TFHE.Sim which generates via TFHE.Sim.Setup simulated shares of the secret key and via TFHE.Sim.Query simulated partial decryptions indistinguishable from the real shares and real partial decryptions. We refer the readers to [FKLS20, BGG+18] for a formal definition of simulation security.

Leveled vs. non-leveled. A TFHE scheme as above is known from LWE; see [BGG+18]. Notice that in the above scheme, d, the depth of the computation, is given as a parameter to the system, and keys and ciphertexts blowup polynomially with d. This is sometimes called a *leveled* scheme. It is known how to get a "non-leveled" scheme, namely, one that works for any (not a priori fixed) d and whose blowup does not depend on d (i.e., it is $\mathsf{poly}(\lambda, \log M)$). Such a system is called a non-leveled threshold FHE and the standard bootstrapping technique [Gen09] can be used to get it. Using this method requires a circular security assumption, that is, the threshold FHE needs to be secure even in the presence of an encryption of the master secret key.

In the statement of our results (Theorems 7.1 and A.1), we shall assume a non-leveled scheme, for simplicity of presentation. However, all of our result can also be stated assuming only a leveled TFHE scheme. In this case, the space blowup will be proportional to $\mathsf{poly}(\lambda, d, \log M)$, where d is the total depth of computation throughout the protocol. Similar blowup and assumptions were required by *all* previous secure MPC compilers [CCLS20, FKLS20] (as well as in many communication-succinct secure computation protocols such as [AJL+12, CM15, MW16]. Nevertheless, we point out that many MPC algorithms have very low depth, usually at most poly-logarithmic in the input size (for example, [ASZ19, ASW18, GU19, HSS19] to name a few). Thus, we expect that using a leveled scheme would suffice in most applications.

3.5 ID-Based Simulation-Extractable zk-SNARKs

In this section we give the definition of ID-based simulation-extractable snarks (idse-zkSNARKs).

Definition 4 (ID-Based Simulation-Extractable zk-SNARKS (idse-zkSNARKs))**.** *An ID-Based simulation-extractable zk-SNARK scheme for the relation R is a tuple of PPT algorithms* $(\Pi.\mathsf{Setup}, \Pi.\mathsf{P}, \Pi.\mathsf{V}, \Pi.\mathsf{Sim}_1, \Pi.\mathsf{Sim}_2, \Pi.\mathsf{Sim}_3)$*, defined as follows:*

- $\Pi.\mathsf{Setup}(1^\lambda, M) \rightarrow crs$: *Takes in the security parameter and a number M and outputs a CRS. We will assume that M is always some* $\mathsf{poly}(\lambda)$.

- $\Pi.\mathsf{P}(crs, \phi, w, \mathsf{id}) \rightarrow \pi$: *takes the crs, the statement, the witness, and an id $\mathsf{id} \in \mathbb{Z}_M$, and outputs a succinct proof π, or $\perp$ if $R(\phi, w) = 0$.*

- $\Pi.\mathsf{V}(crs, \phi, \pi, \mathsf{id}) \rightarrow 0$ *or* 1 : *takes the crs, the statement, the proof, and an id, and outputs* 1 *if the proof verifies correctly with respect to the id.*

- $\Pi.\mathsf{Sim}_1(1^\lambda, M) \to (\widetilde{crs}, \mathsf{td})$: *generates a simulated crs along with master trapdoor* td*.*

- $\Pi.\mathsf{Sim}_2(\widetilde{crs}, \mathsf{td}, \mathsf{id}) \to \mathsf{td}_{\mathsf{id}}$: *Takes as input the crs, the master trapdoor, and an id, and generates a trapdoor for that specific id.*

- $\Pi.\mathsf{Sim}_3(\widetilde{crs}, \mathsf{id}, \mathsf{td}_{\mathsf{id}}, \phi) \to \pi^*$: *takes the simulated crs, an id* id *along with a trapdoor* $\mathsf{td}_{\mathsf{id}}$*, and a statement* ϕ*, and generates a simulated proof* π^* *with respect to* id*.*

We require idse-zkSNARKs to satisfy the following completeness, compactness, adaptive soundness, multi-theorem zero-knowledge, and id-based simulation-extractability requirements.

Definition 5 (Completeness)**.** *A SNARK scheme is said to satisfy completeness if for any* $(\phi, w) \in R$ *and* $\mathsf{id} \in \mathbb{Z}_M$*,*

$$\Pr\left[\Pi.\mathsf{V}(crs, \phi, \pi, \log M) = 1 : \begin{array}{c} crs \leftarrow \Pi.\mathsf{Setup}(1^\lambda, M) \\ \pi \leftarrow \Pi.\mathsf{P}(crs, \phi, w, \mathsf{id}) \end{array}\right] = 1.$$

Definition 6 (Succinctness)**.** *A SNARK scheme must satisfy the following efficiency properties:*

- *The length of any honestly generated setup crs is bounded in size by* $\mathsf{poly}(\lambda)$*.*

- *The length of the proof* π *that* $\Pi.\mathsf{P}(crs, \phi, w, \mathsf{id})$ *outputs, as well as the running time of* $\Pi.\mathsf{V}(crs, \phi, \pi, \mathsf{id})$*, is bounded by*

$$\mathsf{poly}(\lambda + |\phi| + |\mathcal{M}| + \log t + \log M),$$

where $\mathcal{M}$ *is a machine that verifies the relation* R*, and* t *is a bound on the time taken on* $\mathcal{M}$ *for statements of size* $|\phi|$*.*

Definition 7 (Complexity-Preserving Efficiency [BCCT13])**.** *A SNARK scheme is said to be* complexity-preserving *if the running time of* $\Pi.\mathsf{P}(crs, \phi, w, \mathsf{id})$ *is proportional to* $\mathsf{poly}(\lambda) \cdot (|\mathcal{M}| + |\phi| + t + \log M)$*, where* t *is the running time of the verification procedure for* ϕ*, and the space used by* $\Pi.\mathsf{P}(crs, \phi, w, \mathsf{id})$ *is proportional to* $\mathsf{poly}(\lambda) \cdot (|\mathcal{M}| + |\phi| + x + \log M)$*, where* s *is the maximum space used by the verification procedure for* ϕ*.*

We note that [BCCT13] give a generic transformation from any SNARK scheme to a complexity-preserving SNARK scheme. Looking ahead, we note that this transformation preserves the id-based simulation-extractability and zero-knowledge properties.

Definition 8 (Multi-Theorem (Black-Box) Zero-Knowledge)**.** *A SNARK is said to be multi-theorem (black-box) zero knowledge if for all PPT adversaries* $\mathcal{A}$ *and ids* $\mathsf{id} \in \mathbb{Z}_M$*, if* $crs \leftarrow \Pi.\mathsf{Setup}(1^\lambda, M)$ *and* $(\widetilde{crs}, \mathsf{td}) \leftarrow \Pi.\mathsf{Sim}_1(1^\lambda, M)$*,* $\mathsf{td}_{\mathsf{id}} \leftarrow \Pi.\mathsf{Sim}_2(\widetilde{crs}, \mathsf{td}, \mathsf{id})$*, we have that*

$$\left|\Pr\left[\mathcal{A}^{\Pi.\mathsf{P}(crs, \cdot, \cdot, \mathsf{id})}(crs)\right] - \Pr\left[\mathcal{A}^{\mathcal{O}(\widetilde{crs}, \mathsf{id}, \mathsf{td}_{\mathsf{id}}, \cdot, \cdot)}(\widetilde{crs})\right]\right| < \mathsf{negl}(\lambda),$$

where $\mathcal{O}(\widetilde{crs}, \mathsf{id}, \mathsf{td}_{\mathsf{id}}, \phi, w)$ *outputs* $\Pi.\mathsf{Sim}_3(\widetilde{crs}, \mathsf{id}, \mathsf{td}_{\mathsf{id}}, \phi)$ *if* $R(\phi, w) = 1$ *and* $\perp$ *otherwise, and where the probability is taken over the coins of the adversary, the oracle, and the setup algorithms.*

Definition 9 (ID-Based Simulation-Extractability)**.** *A SNARK scheme is said to be id-based simulation-extractable if for all nonuniform PPT adversaries* $\mathcal{A}$ *there exists a non-uniform PPT extractor* $\mathcal{E}_{\mathcal{A}}$ *such that the game* $\mathsf{SimExt}_{\mathcal{A},\mathcal{E}_{\mathcal{A}}}(\lambda)$ *defined below outputs* 1 *with negligible probability in* λ*.*

Game 2 $\mathsf{SimExt}_{\mathcal{A},\mathcal{E}_{\mathcal{A}}}(\lambda)$:

1. Choose uniform random strings $r_{\mathcal{A}}, r_{\mathsf{setup}}$.
2. Initialize $\mathcal{A}$ with security parameter 1^λ and randomness $r_{\mathcal{A}}$.
3. Generate simulated setup parameters and master trapdoor $(\widetilde{crs}, \mathsf{td}) \leftarrow \Pi.\mathsf{Sim}_1(1^\lambda, M; r_{\mathsf{setup}})$, and send $\widetilde{crs}$ to $\mathcal{A}$.
4. Receive corrupted set $\mathcal{C}$ from $\mathcal{A}$.
5. For each $\mathsf{id} \in [M] \setminus \mathcal{C}$, generate $\mathsf{td}_{\mathsf{id}} \leftarrow \Pi.\mathsf{Sim}_2(\widetilde{crs}, \mathsf{td}, \mathsf{id})$, and return the set $\{\mathsf{td}_{\mathsf{id}}\}_{\mathsf{id} \in [M] \setminus \mathcal{C}}$ to $\mathcal{A}$.
6. Receive $(\phi^*, \pi^*, \mathsf{id}^*)$ from $\mathcal{A}$.
7. If $\mathsf{id}^* \notin \mathcal{C}$, then halt and output 0.
8. If $\Pi.\mathsf{V}(\widetilde{crs}, \phi^*, \pi^*, \mathsf{id}^*) = 0$, then halt and output 0.
9. Run the extractor $w^* \leftarrow \mathcal{E}_{\mathcal{A}}(\widetilde{crs}, r_{\mathcal{A}}, \pi_1, \ldots, \pi_t)$ to obtain the witness w^*.
10. If $(\phi^*, w^*) \in R$, output 0. Otherwise, output 1.

The work of [BJPY18] proves the following theorem:[5]

Theorem 3.3. *Assuming the existence of one-way functions and a witness-indistinguishable SNARK for all NP-relations R, there exists an idse-zkSNARK scheme for all NP-relations R.*

3.6 Threshold Signature Scheme with Distributed Reconstruction

In this section, we define a variant of threshold signatures, with a property which we call *distributed reconstruction.* We present the formal definition below. Since for our purposes the only access structure needed is M-out-of-M, we restrict our definition to this case.

Definition 10 (M-out-of-M Threhold Signature Scheme with Distributed Reconstruction (TSDR))**.** *Let $\{P_1, \ldots, P_m\}$ be a set of parties. A M-out-of-M threshold signature scheme with distributed reconstruction is a tuple of algorithms* $(\mathsf{Sig.Setup}, \mathsf{Sig.Sign}, \mathsf{Sig.Combine}, \mathsf{Sig.Verify})$*, defined as follows:*

- $\mathsf{Sig.Setup}(1^\lambda, 1^M)$*: takes as input the security parameter and number of parties M and outputs a pair* $(\mathsf{vk}, \{\mathsf{ssk}_i\}_{i \in [M]})$*, where* vk *is the public verication key and* ssk_i *is P_i's signing key.*
- $\mathsf{Sig.Sign}(\mathsf{ssk}_i, x)$*: outputs a signature $\sigma_{\{i\}}$ for a message x.*
- $\mathsf{Sig.Combine}(\{\sigma_{I_i}\}_i)$*: Assuming $\bigcap_i I_i = \varnothing$, outputs a combined signature for the set $I = \bigcup_i I_i$.*
- $\mathsf{Sig.Verify}_{\mathsf{vk}}(x, \sigma_I)$ *outputs 1 if σ_I is a valid aggregated signature for M on all parties in I, and $I = [M]$.*

We require the threshold signature scheme to satisfy the following correctness, compactness, and unforgeability requirements.

[5]The definition of idse-zkSNARKs given in [BJPY18] has a minor difference from our definition, in that our extractor is given the trapdoors for the honest parties whereas their extractor is not given this information. But the construction and proof in [BJPY18] nonetheless satisfy our definition, since it is strictly weaker than theirs.

Definition 11 (Correctness). *We say a TSDR scheme satisfies correctness if for all λ,*

$$\Pr\left[\mathsf{Sig.Verify}_{\mathsf{vk}}(x, \sigma_{[M]})\right] = 1 - \mathsf{negl}(\lambda)$$

whenever $\sigma_{[M]} \neq \bot$ has been obtained via the interactive algorithm described below.

Game 3 Correctness Interactive Algorithm

1. Generate setup parameters $(\mathsf{vk}, \{\mathsf{ssk}_i\}_{i \in M}) \leftarrow \mathsf{Sig.Setup}(1^\lambda, 1^M)$.
2. Set $\mathcal{S} \leftarrow \varnothing$.
3. Receive message x; for each $i \in [M]$, add $\sigma_{\{i\}} \leftarrow \mathsf{Sig.Sign}_{\mathsf{ssk}_i}(x)$ to $\mathcal{S}$.
4. While the next message received is not "finish":
   (a) Receive $\{I_j\}_j$; if $\bigcup_j I_j \neq \varnothing$, halt and output $\bot$.
   (b) For each I_j, if there is no (I_j, σ_{I_j}) in $\mathcal{S}$, halt and output $\bot$.
   (c) Let $\tilde{I} = \bigcup_j I_j$. Set $\sigma_{\tilde{I}} \leftarrow \mathsf{Sig.Combine}(\{\sigma_{I_j}\}_j)$. Add $\sigma_{\tilde{I}}$ to $\mathcal{S}$.
5. If no pair $([m], \sigma_{[M]})$ is in $\mathcal{S}$, halt and output $\bot$.
6. Otherwise, output $\sigma_{[M]}$.

Definition 12 (Compactness). *We say a TSDR scheme satisfies compactness if there exists a polynomial* poly *such that for all λ and m, the following holds:*

- vk *and every* ssk_i *produced by* $\mathsf{Sig.Setup}(1^\lambda, 1^M)$ *have size bounded by* $\mathsf{poly}(\lambda)$.
- *Assuming* $(\mathsf{vk}, \{\mathsf{ssk}_i\})$ *were produced by* $\mathsf{Sig.Setup}$ *as in the first bullet, the algorithms* $\mathsf{Sig.Sign}_{\mathsf{ssk}}(x)$ *and* $\mathsf{Sig.Verify}_{vk}(x, \sigma_I)$ *take maximum space* $\mathsf{poly}(\lambda) \cdot |x|$.
- *Assuming* $(\mathsf{vk}, \{\mathsf{ssk}_i\})$ *were produced by* $\mathsf{Sig.Setup}$ *as in the first bullet and all σ_{I_i} were produced by* $\mathsf{Sig.Setup}$ *as in the second bullet or a recursive application of* $\mathsf{Sig.Combine}$, $\mathsf{Sig.Combine}(\{\sigma_{I_i}\}_i)$ *takes maximum space* $\mathsf{poly}(\lambda) \cdot |\{\sigma_{I_i}\}_i| \cdot |x|$ *and outputs a signature of size bounded by* $\mathsf{poly}(\lambda) \cdot |x|$.
- *Assuming* $(\mathsf{vk}, \{\mathsf{ssk}_i\})$ *were produced by* $\mathsf{Sig.Setup}$ *as in the first bullet and all σ_{I_i} were produced by* $\mathsf{Sig.Setup}$ *as in the second bullet or a recursive application of* $\mathsf{Sig.Combine}$, $\mathsf{Sig.Verify}_{\mathsf{vk}}(x, \sigma_I)$ *takes maximum space* $\mathsf{poly}(\lambda) \cdot |x|$.

Definition 13 (M-out-of-M Unforgeability). *We say a TSDR scheme satisfies M-out-of-M unforgeability if for any PPT adversary $\mathcal{A}$, the experiment* $\mathsf{Unforgeability}_{\mathcal{A}}$ *defined below outputs* 1 *with negligible probability in λ.*

Game 4 $\mathsf{Unforgeability}_{\mathcal{A}}(\lambda, M)$:

1. Generate setup parameters $(\mathsf{vk}, \{\mathsf{ssk}_i\}_{i \in M}) \leftarrow \mathsf{Sig.Setup}(1^\lambda, 1^M)$.
2. Initialize $\mathcal{A}$ with security parameter 1^λ and send vk to $\mathcal{A}$.
3. Receive the set $\mathcal{C} \subsetneq [M]$ of corrupted parties from $\mathcal{A}$, and reply with $\{\mathsf{ssk}_i\}_{i \in \mathcal{C}}$.

4. Set $\mathcal{S} \leftarrow \varnothing$.
5. For each query (I, x) received from $\mathcal{A}$:
   (a) Compute the signature $\sigma_i \leftarrow \mathsf{Sig.Sign}_{\mathsf{ssk}_i}(x)$ for each $i \in I$.
   (b) Send $\{\sigma_i\}_{i \in I}$ to $\mathcal{A}$.
   (c) If there is a set I_x such that $(x, I_x) \in \mathcal{S}$, replace this tuple in $\mathcal{S}$ with $(x_r, I_x \cup I)$. Otherwise, add $(x_r, \mathcal{C} \cup I)$ to $\mathcal{S}$.
6. Receive (x^*, σ^*) from $\mathcal{A}$. If $(x^*, [M]) \notin \mathcal{S}$ and $\mathsf{Sig.Verify}_{\mathsf{vk}}(x^*, \sigma^*) = 1$, then output 1. Otherwise output 0.

We now import the following theorem from [BGG$^+$18], which proves that there exists a TSDR scheme assuming hardness of LWE.

Theorem 3.4 (Theorems 8.17 to 8.22 in [BGG$^+$18])**.** *Assuming hardness of LWE, there exists a threshold signature scheme with distributed reconstruction satisfying the correctness, compactness, and unforgeability requirements in Definitions 11 to 13.*

4 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 4.1. The security of MPC algorithms is defined in a standard way, following the security definition in multiparty 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 4.2 and 4.3, respectively).

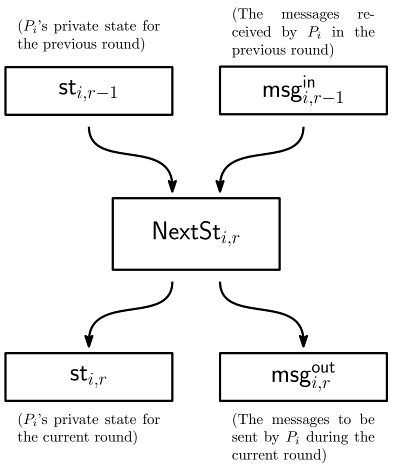
4.1 The Massively Parallel Computation Model

We briefly recall the massively parallel computation (MPC) model, following Chan et al. [CCLS20] 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 $r \in [R]$. See Figure 1 on page 20.
- 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. [CCLS20] 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.



(P_i's private state for the previous round)
(The messages received by P_i in the previous round)
$\mathsf{st}_{i,r-1}$
$\mathsf{msg}^{\mathsf{in}}_{i,r-1}$
$\mathsf{NextSt}_{i,r}$
$\mathsf{st}_{i,r}$
$\mathsf{msg}^{\mathsf{out}}_{i,r}$
(P_i's private state for the current round)
(The messages to be sent by P_i during the current round)

Figure 1: Input/output for NextSt (setup omitted for clarity)

4.2 Malicious Security for MPC protocols

We now define malicious security for MPC protocols following the general real-ideal framework (given e.g., in [Gol09]) 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 14. *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 14 must not have access to this oracle.*

Alternatively, sometimes a non-programmable variant is used. Specifically, here the distinguisher in Definition 14 does *have access to this oracle and so* $\mathcal{S}$ *cannot program answers to its choice.*

4.3 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 [AJL+12], 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 4.2. 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 15. *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* $\mathsf{ideal}^{\mathcal{F}f}_{\mathcal{S}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$.

5 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 5.1. *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 [HW15] 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 Theorem 5.2, and Theorem 5.1 is a direct corollary of it. The proof is an adaptation of [HW15] and is given for completeness.

Theorem 5.2. *Assume that a pseudorandom function exists. Let* COMPILER *be a compiler that takes as input any M-machine protocol Π and outputs another protocol $\tilde{\Pi}$ that has the same input-output functionality and is also* P2P-semi-malicious secure*. The compiler may assume a PKI or a (non-programmable) random oracle. It must holds that $R \in \Omega(M/S)$, where R is the round complexity of $\tilde{\Pi}$ and S is the space consumed by each machine.*

Recall that in the MPC model it is often the case that R is tiny and that S is some small polynomial in M, for instance $R \in O(1)$ and $S = M^{0.1}$. The above theorem says that it is impossible to generically compile protocols to P2P-semi-malicious counterparts without significantly increasing either the round complexity or the space complexity. Notice that the above theorem only holds for compiler for all MPC functionalities and does not work if we only care about "short output" protocols, as we exemplify in Appendix A. Additionally, it does not apply if we only need semi-honest security, which was achieved in [FKLS20].

Required definitions. As mentioned, the contradiction is obtained by showing that a secure compiler can be used to compress the outputs of a PRF. The latter do not have (Shannon) entropy because they only appear to be random for computationally bounded attackers. Still, they do have a computational variant of entropy which we want to use. Specifically, the notion that we use is called "Yao incompressibility entropy"[Yao82] which, roughly, captures the fact that a given distribution is *computationally* close to another distribution which does have (Shannon) entropy. For technical reasons, we actually need a conditional variant which we define next. Below, we give the definition adapted to our (multiparty) setting.

Definition 16 (Conditional Yao Incompressibility Entropy [HW15])**.** *Let $k = k(\lambda)$ be an integer-valued function of security parameter λ. A probability ensembles $X_{1,\lambda}, \ldots, X_{n,\lambda}$ has* Yao incompressibility entropy *at least k conditioned on $Z_{1,\lambda}, \ldots, Z_{m,\lambda}$, denoted $H^{Yao}(X_{1,\lambda}, \ldots, X_{n,\lambda} \mid Z_{1,\lambda}, \ldots, Z_{m,\lambda}) \geq k$, if for every pair of circuit ensembles C_λ, D_λ (called "compressor" and "decompressor") of size* $\mathsf{poly}(\lambda)$ *where C_λ has output-size at most $k(\lambda) - 1$, it holds that*

$$\Pr[D_\lambda(C_\lambda(x_1, \ldots, x_n, z_1, \ldots, z_m)) = (z_1, \ldots, z_m)] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where $(x_1, \ldots, x_n, z_1, \ldots, z_m) \leftarrow (X_{1,\lambda}, \ldots, X_{n,\lambda}, Z_{1,\lambda}, \ldots, Z_{m,\lambda})$.

Definition 17 (Yao Incompressibility Entropy of Function [HW15])**.** *We say that a function $f : \{0,1\}^{\ell_1} \times \ldots \times \{0,1\}^{\ell_M} \to \{0,1\}^{L_2} \times \ldots \times \{0,1\}^{L_M}$ has* Yao incompressibility entropy *at least*

k, denoted $H^{Yao}(f) \geq k$, if there exist a probability ensemble $X_1, \ldots, X_M$ of distributions over $\{0,1\}^{\ell_1}, \ldots, \{0,1\}^{\ell_M}$ such that $H^{Yao}(f(X_1, \ldots, X_M) \mid X_2, \ldots, X_M) \geq k$.

Lemma 5.1. *Let $f : \{0,1\}^{\ell_1} \times \ldots \times \{0,1\}^{\ell_M} \to \{0,1\}^{L_2} \times \ldots \times \{0,1\}^{L_M}$ be a functionality, and let Π be a protocol for evaluating f with P2P-semi-malicious security (assuming a PKI and a non-programmable random oracle against an adversary that control $P_2, \ldots, P_M$. If $H^{Yao}(f) \geq k$, then the outgoing communication from P_1 (throughout the protocol) must be at least k bits.*

Proof. Assume for contradiction that $H^{Yao}(f) \geq k$, but the outgoing communication from P_1 is at most $k - 1$ bits. Since Π securely evaluates f against semi-malicious adversaries that control $P_2, \ldots, P_M$, for every such adversary $\mathcal{A}$, there exists an efficient simulator $\mathcal{S}_{\mathcal{A}}$ that satisfies Definition 15. For every $\rho = \{r_i\}_{i \in \{2,\ldots,M\}}$, define the adversary $\mathcal{A}_\rho$ that follows the protocol's specification except that it uses randomness r_i to implement machine i's messages. For each such ρ, let $\mathcal{S}_{\mathcal{A}_\rho}$ be the corresponding simulator. There exists ρ for which $\mathcal{S}_{\mathcal{A}_\rho}$ does not abort.

We use $\mathcal{S}_{\mathcal{A}_\rho}$ to build an efficient compression $\mathcal{C}_\rho$ and decompression $\mathcal{D}_\rho$ procedures such that the compressor manages to "compress" a sample from $f(X_1, \ldots, X_M) \mid X_2, \ldots, X_M$ with only $k-1$ bits which the decoder can then use to recover the original sample. This, in turn, implies a contradiction to the assumption that $H^{Yao}(f) \geq k$. If setup is used, the compressor and decompressor know it as public information. Since the setup (either PKI or non-programmable random oracle) is independent of the X_i's, the below argument works in the same way. Specifically, the compressor and decompressor also know the PKI and provide it to the simulator or alternatively they have access to the random oracle and they use it to answer the parties' / simulator's queries. For simplicity of description we ignore the setup.

Let $X_1, \ldots, X_M$ be distributions of parties inputs that maximize $H^{Yao}(f(X_1, \ldots, X_M) \mid X_2, \ldots, X_M)$ so that $H^{Yao}(f) \geq k$. For an input $\{x_2, \ldots, x_M, f(x_1, \ldots, x_M)\}$ (from the support of $(X_2, \ldots, X_M, f(X_1, \ldots, X_M)\})$), the compressor $\mathcal{C}_\rho$ does the following:

1. Run $\mathcal{S}_{\mathcal{A}_\rho}(1^\lambda, \{P_2, \ldots, P_M\}, \{x_i, r_i\}_{i \in \{2,\ldots,M\}}, f(x_1, \ldots, x_M))$ to obtain $\mathsf{view}_{\mathcal{C}}^{\mathcal{S}_{\mathcal{A}_\rho}}$.

2. Extract from $\mathsf{view}_{\mathcal{C}}^{\mathcal{S}_{\mathcal{A}_\rho}}$ the messages $\{m_{1,i}^j\}_{i \in \{2,\ldots,M\}, j \in [R]}$ sent by party P_1, where $m_{1,i}^j$ is the message from P_1 to P_i at round j.

3. Output $\{m_{1,i}^j\}_{i \in \{2,\ldots,M\}, j \in [R]}$

The decompressor $\mathcal{D}_\rho$ do the following:

1. Execute the protocol Π (in the head) using ρ, λ and inputs $\{x_i\}_{i \in \{2,\ldots,M\}}$ and use the messages from the compressor $\mathcal{C}_\rho$, as P_1 to simulate the messages from P_1. Denote $y_2', \ldots, y_M'$ the output of Π.

2. Output $y_2', \ldots, y_M'$.

By correctness of $\mathcal{S}_{\mathcal{A}_\rho}$, we know that $\Pr[y_2', \ldots, y_M' = f(x_1, \ldots, x_M)] \geq 1 - \mathsf{negl}(\lambda)$. Hence, for large enough λ, with very high probability, we successfully compressed and decompressed $k(\lambda)$ bits of information using only at most $k(\lambda) - 1$. This contradicts $H^{Yao}(f) \geq k$. □

Proof of Theorem 5.2. Using a PRF we design an M-machine protocol that realizes a functionality f for which $H^{Yao}(f) = M - 1$. In our functionality, all parties are input-less except P_1 which holds a randomly chosen PRF key k of length λ (chosen independently of the setup). The output of each party P_i for $i \in \{2, \ldots, M\}$ is the evaluation of the PRF function at point i.

There is a trivial insecure implementation of this functionality in the MPC model: The parties invoke $\mathsf{Distribute}_{\sqrt{S}}(k)$ so that all parties receive k, the PRF key, and locally evaluate the PRF at their respective index. However, as we shall see, there is no way to realize this functionality with P2P-semi-malicious security.

Indeed, since the setup is independent of the private inputs (in our case the PRF key of party P_1), and since the PRF outputs are computationally indistinguishable from random, we directly obtain $H^{Yao}(f(k)) = M - 1$. By Lemma 5.1, the total outgoing communication from party P_1 is $\Omega(M)$. This can only hold if $R \in \Omega(M/S)$, as needed.

□

6 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.

6.1 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 $\gamma^k$.
2: For each round $k \in [t]$, all the parties on level $t+1-k$ send $x$ to the parties on level $t-k$.
```

6.2 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 $\gamma^k$.
   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 $x_{1,t} = x_1 \, \mathsf{op} \ldots \mathsf{op} \, x_M$.
```

7 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 7.1 (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 as in Section 3.4.*

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.

The rest of this section is devoted to proving Theorem 7.1. The protocol followed by its efficiency analysis are given in Section 7.1. The security proof is given in Section 7.2.

7.1 The Protocol

We assume a PKI, so every party P_i knows the public key along with its secret key sk_i. At a high level, the protocol is divided into two main phases, as in the short output protocol, with the major differences occurring in the second phase. In the first phase, as in the short output protocol, each party encrypts its initial state under pk, and the parties carry out an encrypted version of the original (insecure) MPC protocol using the TFHE evaluation function. In the second phase, the parties interact with each other so that all parties obtain an obfuscation of a circuit which will allow them to decrypt their outputs and nothing else. This involves carrying out a sub-protocol $\mathsf{CalcSSBHash}$ in which the parties collectively compute a somewhere-statistically-binding (SSB) commitment to their ciphertexts. Recall that an SSB hash has a Merkle-tree structure which is designed specifically to enable security proofs when using iO.

$\mathsf{CalcSSBHash}$. The purpose of this protocol is for all parties to know an SSB commitment z to their collective inputs, and for each party P_i to know an opening π_i for its respective input. We will perform this process over a tree with arity γ, mirroring the Merkle-like tree of the SSB hash. In the first round, the parties use $\mathsf{SSB.Start}$, and then send the resulting label to the parties $P_{i'}$, $i' \equiv 0 \pmod{\gamma}$ (call these nodes the parents). Each of these parties $P_{i'}$ then uses $\mathsf{SSB.Combine}$ on the labels $\{y_{i,0}\}$ of its children to get a new combined label $y_{i',1}$, and then all the $P_{i'}$ parties send their new labels to $P_{i''}$, $i'' \equiv 0 \pmod{\gamma^2}$. In addition, since the string each party P'_i now has a part of its children's openings, namely $y_{i',1}$ and the set $\{y_{i,0}\}$ of sibling labels, it sends $\pi_{i,1} = (y_{i',1}, \{y_{i,0}\})$ to each of its children to be used as openings.

This 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 $\pi_{i,j}$ received from its parent to all its children. At the end, all parties will know z and π_i.

The formal description of the protocol is below. Note that we use the subprotocols $\mathsf{Distribute}$ and $\mathsf{CalcSSBHash}$; $\mathsf{Distribute}$ was defined in the previous section, and $\mathsf{CalcSSBHash}$ is defined after the main protocol.

P2P Semi-Malicious Compiler for long output Protocols

Input: Party P_i has input x_i to the underlying MPC protocol and circuits $\mathsf{NextSt}_{i,1}, \ldots, \mathsf{NextSt}_{i,R}$, as described in Section 4.1.

Output: In the end of the protocol, each party P_i will receive the output $y_i \in \{0,1\}^{l_{\mathsf{out}}}$, where $(y_1, \ldots, y_M) = f(x_1, \ldots, x_M)$ and f is the functionality that the original protocol Π computes.

1: **Setup Phase:** Each party P_i knows the security parameter λ, the public keys pk, pk^* along with a share of the master secret keys, sk_i, sk_i^*, respectively, where $(pk, \{sk_i\}_{i\in[M]}), (pk^*, \{sk_i^*\}_{i\in[M]}) \leftarrow \mathsf{TFHE.Setup}(1^\lambda, M)$.

2: **Encrypted MPC Phase:** For the first R rounds, the behavior of each party P_i is exactly as in the encrypted MPC phase of the short output protocol. Specifically,

(a) Each party P_i encrypts its input x_i using the public key pk: $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}(\mathsf{pk}, x_i)$.

(b) Parties evaluate the underlying protocol round by round. During round $r \in [R]$, each party P_i computes $ct_{\mathsf{st}_{i,r}} || ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}} \leftarrow \mathsf{TFHE.Eval}(\mathsf{NextSt}_{i,r}, ct_{\mathsf{st}_{i,r-1}} || ct_{\mathsf{msg}^{\mathsf{in}}_{i,r-1}})$ and sends encrypted messages $ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}}$ to other parties according to the original protocol Π.

(c) In the end of the evaluation phase, each party P_i holds $ct_{y_i} = ct_{\mathsf{st}i,R}$, the encryption of the output y_i.

3: **Output Padding Phase:** After the R rounds of the encrypted MPC protocol are done, each party P_i does the following:

(a) Compute a random string $pad_i \leftarrow \{0,1\}^{l_{out}}$.

(b) Calculate $ct_{pad_i} \leftarrow \mathsf{TFHE.Enc}_{pk}(pad_i)$.

(c) Calculate $ct_{o,i} \leftarrow \mathsf{TFHE.Eval}(\oplus, ct_{pad_i}, ct_{y_i})$, the TFHE evaluation of the circuit which pads the output of the i party y_i with the corresponding pad pad_i.

4: **Output Circuit Generation Phase:** At the end of the previous phase, each party P_i has an encryption $ct_{o,i}$ of their output padded with pad_i. The parties then coordinate with each other in a manner which is now described, so that at the end P_1 has an obfuscation of the circuit $C_{h,z_y,z_r,sk}$, defined below.

- **SSB Hash phase:**

  (a) Each party chooses a uniform random string $r_{ssb,i}$, and the parties run the (semi-malicious) short output compiler with $(pk^*, \{sk_i^*\}_{i\in[M]})$ over the protocol $\mathsf{Combine}_\lambda(\oplus, \{r_{ssb,i}\}_{i\in[M]})$, so that P_1 output is $r_{ssb} = r_{ssb,1} \oplus \ldots \oplus r_{ssb,M}$.

  (b) P_1 generates an SSB hash key $h \leftarrow \mathsf{SSB.Setup}(1^\lambda, 2 \cdot l_{out}, \lceil \log_\lambda M \rceil, \lambda, 1; r_{ssb})$ with $2 \cdot l_{out}$ as the block size and 1 as the statistically binding index.

  (c) The parties run the protocol $\mathsf{Distribute}_\lambda(h)$.

  (d) The parties run the protocol $\mathsf{CalcSSBHash}_{h,\lambda}(\{ct_{o,i}\}_{i\in[M]})$ defined below, so that each party P_i obtains an SSB commitment z_y and an opening $\pi_{y,i}$ to $ct_{o,i}$.

- **Randomness phase:**

(e) Each party chooses a uniform random string $r_{seed,i}$, and the parties run the (semi-malicious) short output compiler with $(pk^*, \{sk_i^*\}_{i\in[M]})$ over the protocol $\mathsf{Combine}_\lambda(\oplus, \{r_{seed,i}\}_{i\in[M]})$, so that P_1 output is $r_{seed} = r_{seed,1} \oplus \ldots \oplus r_{seed,M}$.

(f) The parties run the protocol $\mathsf{Distribute}_\lambda(r_{seed})$.

(g) Each party P_i sends a query to random oracle to and sets $r_{o,i} = \mathcal{O}(r_{seed}\|i)$ (which is of size l_{out}).

(h) Each party P_i calculate offline (locally, without communicate with other parties) $\mathsf{CalcSSBHash}_{h,\lambda}(\{\mathcal{O}(r_{seed}\|i)\}_{i\in[M]})$ defined below, to obtain an SSB commitment z_r, and store only the opening $\pi_{r,i}$, that related to his index i.

- **Circuit Generation phase:**

(i) Each party chooses a uniform random string $r_{iO,i}$, and the parties run the short output compiler with $(pk^*, \{sk_i^*\}_{i\in[M]})$ over the protocol $\mathsf{GenerateCircuit}_{h,z_y,z_r,\lambda}(\{(sk_i, r_{iO,i})\}_{i\in[M]})$ defined below, so that P_1 obtains an obfuscation C' of the circuit $C_{h,z_y,z_r,sk}$, also defined below.

(j) The parties run $\mathsf{Distribute}(C')$.

5: **Offline Output Decryption Phase:** Once every party knows C', each party P_i can run $C'(i, ct_{o,i}, r_{o,i}, \pi_{y,i}, \pi_{r,i})$ to obtain y_i', P_i's padded output under the original MPC protocol. P_i can then compute $y_i \leftarrow y_i \oplus pad_i$.

$\mathsf{CalcSSBHash}_{h,\gamma}(\{x_i\}_{i\in[M]})$:

Input: Each party P_i has a key h and x_i. In this protocol we will number the parties starting at 0 (so the first party will be P_0).

Output: The protocol stops after $2\lceil\log_\gamma M\rceil$ rounds, and every party P_i knows the SSB tree root y and the opening π_i of x_i.

1: **Parameters:** Let $\lambda \leq S$. Assume h is an SSB hash which has been initialized with γ and $t = \lceil\log_\gamma M\rceil$.

2: **Before starting:** Each party P_i first computes $\leftarrow \mathsf{SSB.Start}(h, x_i)$ to obtain a string $y_{i,0}$ of size λ.

When carrying out the protocol, we will divide the parties into subsets. Let $S_r = \{P_i \mid i \equiv 0 \pmod{\gamma^r}\}$ (and let $S_0 = \{P_i\}_{i\in[M]}$), let the set of children for i in S_r be $D_{i,r} = \{P_j \mid j \equiv 0 \pmod{\gamma^{r-1}} \text{ and } i \leq j \leq i + \gamma^r\}$, and let the parent of i in S_r be $q_{i,r} = \gamma^r\lfloor i/\gamma^r\rfloor$.

3: **Round k for $k = 1, \ldots, \lceil\log_\gamma M\rceil + 1$:**
Parties in the sets S_{k-t}, $t = 1, 3, \ldots, 2\lceil k/2\rceil - 1$ will participate.

- Each party P_i in S_{k-1} does the following:

(a) If $k - 1 > 0$, receive $y_{j,k-2}$ from each $P_j \in D_{i,k-1}$.

(b) If $k - 1 > 0$, calculate $y_{i,k-1} \leftarrow \mathsf{SSB.Combine}(h, \{y_{j,k-2}\}_{P_j\in D_{i,k-1}})$, and send $(y_{i,k-1}, \{y_{j,k-2}\}_{j\in D_{i,k-1}}$ to all parties P_j, $j \in D_{i,k-1}$.

(c) Send $y_{i,k-1}$ to P_{q_i}.

- Each party P_j in S_r for $r = 0, \ldots, k-2$ does the following:

(a) Check if received $\pi_{j,r'} = (y_{i,r'}, \{y_{j',r'-1}\}_{j'\in D_{i,r'}})$ from P_{q_j}.

(b) If so, append $\pi_{j,r'}$ to π_j.

(c) If $r > 0$, send $\pi_{j,r'}$ to all $P_{j''} \in D_{j,r}$.

4: **Round k' for $k' = \lceil \log_\gamma M \rceil + 2, \ldots, 2\lceil \log_\gamma M \rceil + 1$:**
Each party P_j in S_r for $r = 0, \ldots, \lceil \log_\gamma M \rceil$ does the following:

(a) Check if received $\pi_{j,r'} = (y_{i,r'}, \{y_{j',r'-1}\}_{j' \in D_{i,r'}})$ from P_{q_j}.

(b) If so, append $\pi_{j,r'}$ to π_j.

(c) If $r > 0$, send $\pi_{j,r'}$ to all $P_{j''} \in D_{j,r}$.

$\mathsf{GenerateCircuit}_{h,z_y,z_r,\gamma}(\{(sk_i, r_{iO,i})\}_{i \in [M]})$:

Input: P_1 the SSB commitment z; each party P_i has sk_i.

1. Parties run $\mathsf{Combine}_\gamma(+, \{sk_i\})$ so that P_1 has the master secret key sk.
2. Parties run $\mathsf{Combine}_\gamma(+, \{r_{iO,i}\})$ so that P_1 has $r_{iO} = \sum r_{iO,i}$.
3. P_1 calculates the obfuscation $C' \leftarrow iO(C_{h,z_y,z_r,sk}; r_{iO})$.

Output: At the end of the protocol, P_1's output is defined as C'. All other parties have blank output.

Circuit $C_{h,z_y,z_r,sk}(i, ct, r, \pi_y, \pi_r)$:

1. If $\mathsf{SSB.Verify}(h, z_y, i, ct, \pi_y) = 1$, and $\mathsf{SSB.Verify}(h, z_r, i, r, \pi_r) = 1$:
   (a) Output $\mathsf{TFHE.Dec}_{sk}(ct)$.
2. Otherwise, output $\perp$.

Correctness. Correctness of the long output protocol follows from correctness of the TFHE scheme, the iO scheme, the short output protocol, the SSB hash, and the $\mathsf{CalcSSBHash}$ subprotocol. To see why correctness holds for the $\mathsf{CalcSSBHash}$ subprotocol, recall that the end of the protocol each party P_i should know the root label z of the tree along with an opening π_i corresponding to P_i's input x_i. Note that each P_i receives messages for each ancestor of the leaf node $y_{i,0}$ corresponding to P_i's input, and each message is of the form $(y_{i',r}, \{y_{j,r-1}\}_{j \in D_{i',r}})$, containing the ancestor's label and the ancestor's direct children's labels. This is sufficient to learn the root label z of the tree as well as all node labels along the path to $y_{i,0}$ along with the labels of the siblings of all nodes along this path. From this, P_i can onstruct π_i.

Efficiency. Note that, since $M \in \mathsf{poly}(\lambda)$ that mean there exists some constant $\epsilon > 0$, such that $\lambda \leq M^{\epsilon}$, so $\lceil \log_{\lambda} M \rceil \leq \lceil \log_{M^{\epsilon}} M \rceil = \lceil \epsilon^{-1} \rceil = O(1)$.

The encrypted MPC and output padding phases of the long output protocol takes exactly R rounds. The output circuit generation phase consists of three executions of $\mathsf{Distribute}_{\lambda}$, a short output protocol for $\mathsf{Combine}_{\lambda}$ and $\mathsf{GenerateCircuit}$, and the $\mathsf{CalcSSBHash}$ protocol, all of those protocols takes $O(\log_{\lambda} M) = O(1)$ rounds. It follows that the total number of rounds used by the long output protocol is $R + O(1)$.

The maximum additional space used by each party during $\mathsf{CalcSSBHash}$ is $O((\log_{\lambda} M + \lambda) \cdot \lambda) = \mathsf{poly}(\lambda)$, The local calculation of z_r and $\pi_{r,i}$ can be performed by store at most $O(\lambda)$ ciphertexts of each merkle-tree depth at any time, so the additional space is $O(\log_{\lambda} M \cdot \lambda) = \mathsf{poly}(\lambda)$ By the space bounds on the short output protocol and the $\mathsf{Distribute}_{\lambda}$ protocol, the total space used in the long output protocol is $S \cdot \mathsf{poly}(\lambda)$.

Note 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 $\mathsf{poly}(\lambda)$. It follows that our compiler also preserve the communication, with same multiplicative blowup like in space.

7.2 Proof of Security

To prove security, for every P2P semi-malicious adversary, we exhibit a simulator for the protocol given above. This simulator will generate a view of an arbitrary set of corrupted parties which will be indistinguishable from the view of the corrupted parties in a real-world execution of the protocol. Note that the simulator receives the public key which is assumed to be generated honestly by the TFHE setup algorithm, and also receives the set of corrupted parties $\mathcal{C}$ as input. This allows the corrupted set $\mathcal{C}$ to be chosen based on the public key.

The behavior of the simulator is described below.

Long Output Simulator

Input: The simulator receives the corrupted set $\mathcal{C}$, the public key pk, the corrupted parties' inputs $\{x_i\}_{i \in \mathcal{C}}$, and the outputs $\{y_i\}_{i \in \mathcal{C}}$.

Output: In the end of the protocol, each corrupted party P_i will receive the output y_i where $(y_1, \ldots, y_M) = f(x_1, \ldots, x_M)$ and f is the functionality that the original protocol Π computes.

1: **Simulated Setup:**

- To generate the corrupted parties' secret keys, the simulator uses the TFHE simulated setup: $(\{sk_i\}_{i \in \mathcal{C}}, \sigma_{sim}) \leftarrow \mathsf{TFHE.Sim.Setup}(pk, \mathcal{C})$, $(\{sk_i^*\}_{i \in \mathcal{C}}, \sigma_{sim}^*) \leftarrow \mathsf{TFHE.Sim.Setup}(pk^*, \mathcal{C})$, and sends $\{sk_i, sk_i^*\}_{i \in \mathcal{C}}$ to the adversary.
- Throughout the execution, upon every random oracle query, the simulator records it in a data structure we denote *cache*. If the given query is already present in *cache*, the simulator returns the previous answer. Otherwise, if the query is new, it samples a uniform sequence of bits, records them for future queries, and responds with them. That is,

$$\mathcal{O}(x) = \begin{cases} cache(x) & x \in cache \\ \text{uniform random string} & otherwise \end{cases}$$

2: **Simulated Encrypted MPC Phase:** The simulator performs this phase in the same way as was done in the short output simulator. Specifically:

(a) For each honest party P_i, the simulator computes an encryption of 0: $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{|x_i|})$

(b) The simulator executes the underlying protocol round by round with the adversary. During round $r \in [R]$, each party P_i computes $ct_{\mathsf{st}_{i,r}} || ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}} \leftarrow \mathsf{TFHE.Eval}(\mathsf{NextSt}_{i,r}, ct_{\mathsf{st}_{i,r-1}} || ct_{\mathsf{msg}^{\mathsf{in}}_{i,r-1}})$ and sends encrypted messages $ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}}$ to other parties according to the original protocol Π.

(c) In the end of the evaluation phase, each party P_i holds $ct_{y_i} := ct_{\mathsf{st}_{i,R}}$.

3: **Simulated Output Padding Phase:** After the R rounds of the encrypted MPC protocol are done, the simulator does the following on behalf of each honest party P_i:

(a) Calculate $ct_{pad_i} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{l_{out}})$.

(b) Calculate $ct_{o,i} \leftarrow \mathsf{TFHE.Eval}(\oplus, ct_{pad_i}, ct_{y_i})$.

4: **Simulated Output Circuit Generation Phase:** At the end of the encrypted execution of the MPC protocol, each party P_i has an encryption $ct_{o,i}$ of their output, The simulator then simulates the output circuit generation phase in the following manner, so that at the end P_1 has an obfuscation of the circuit $\tilde{C}_{h,z_y,z_r,sk}$, defined below.

- **SSB Hash phase:**

(a) The simulator uses the short output simulator for the compiled $\mathsf{Combine}_\lambda$ protocol, where the protocol output is set to be uniform random string r_{ssb}, so P_1 holds r_{ssb}.

(b) If $1 \notin \mathcal{C}$, the simulator runs $h \leftarrow \mathsf{SSB.Setup}$, for P_1. so he holds h.

(c) The simulator runs the protocol $\mathsf{Distribute}_\lambda(h)$ with the adversary like in the real world execution.

(d) The simulator runs the protocol $\mathsf{CalcSSBHash}_{h,\lambda}(\{ct_{o,i}\}_{i\in[M]})$ with the adversary like in the real world execution.

- **Program the Random Oracle:**

(a) The simulator choose uniform random string r_{seed}, and PRF key k.

(b) The simulator use the adversary's witness tape to obtain $\{pad_i\}_{i\in\mathcal{C}}$.

(c) From this point the simulator will answer to the random oracle queries as follows:

$$\mathcal{O}(x) = \begin{cases} PRF_k(i) \oplus y_i \oplus pad_i & x \notin cache \wedge x \in \{r_{seed}\|i\}_{i\in\mathcal{C}} \\ cache(x) & x \in cache \wedge x \notin \{r_{seed}\|i\}_{i\in\mathcal{C}} \\ \perp \text{ (and abort)} & x \in cache \wedge x \in \{r_{seed}\|i\}_{i\in\mathcal{C}} \\ \text{uniform random string} & otherwise \end{cases}$$

- **Randomness phase:**

(a) The simulator uses the short output simulator for the compiled $\mathsf{Combine}_\lambda$ protocol, where the protocol output is set to be r_{seed}, so P_1 holds r_{seed}.

(b) The simulator runs the protocol $\mathsf{Distribute}_\lambda(r_{seed})$ with the adversary as in the real world execution.

(c) The simulator set $r_{o,i} = \mathcal{O}(r_{seed}\|i)$ for each honest party P_i.

(d) The simulator runs the protocol $\mathsf{CalcSSBHash}_{h,\lambda}(\{\mathcal{O}(r_{seed}\|i)\}_{i\in[M]}\}_{i\in[M]})$ locally to obtain z_r.

- **Circuit Generation phase:**

(a) The simulator choose uniform random string $r_{iO,i}$ for each honest party P_i, and uses the short output simulator for the compiled $\mathsf{GenerateCircuit}$ protocol, where the protocol output is set to be the obfuscation $\tilde{C}' = iO(\tilde{C}_{h,z_y,z_r,k})$.

(b) The simulator runs the protocol $\mathsf{Distribute}_\lambda(\tilde{C}')$ with the adversary like in the real world execution.

Circuit $\tilde{C}_{h,z_y,z_r,k}(i, ct, r, \pi_y, \pi_r)$:

```
1. If $\mathsf{SSB.Verify}(h, z_y, i, ct, \pi_y) = 1$, and $\mathsf{SSB.Verify}(h, z_r, i, r, \pi_r) = 1$:
    (a) Output $r \oplus PRF_k(i)$.
2. Otherwise, output $\perp$.
```

Claim 1. *Assuming security of the TFHE scheme, security of the SSB hash function, security of the puncturable PRF, and security of the iO scheme, the output of the simulator is indistinguishable from the view of a semi-malicious adversary which corrupts all parties in $\mathcal{C}$.*

We prove the claim via a sequence of hybrids, described below.

- **Hybrid H_0:** In this hybrid, the simulator behaves identically to the real world, setting the corrupted parties' secret keys to be the ones generated by the TFHE setup, and running the real-world protocol.
- **Hybrid H_1:** In this hybrid, the simulator behaves in the same way as H_0, except that it uses the TFHE simulator to choose the corrupted parties' secret keys, and it simulate the random oracle as described in the simulated setup phase.
- **Hybrid H_2:** In this hybrid, the simulator behaves in the same way as H_1, except that it uses the short output protocol simulator to simulate the executions of $\mathsf{Combine}_\lambda$ runs.
- **Hybrid H_3:** In this hybrid, the simulator behaves in the same way as in H_2, except that it samples a PRF key k, programs the random oracle, and uses the short output protocol simulator to simulate the executions of $\mathsf{GenerateCircuit}$.
- **Hybrid H_4:** In this hybrid, the simulator behaves in the same way as in H_3, except that it sets $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{|x_i|})$ and $ct_{pad_i} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{l_{out}})$ for each honest party P_i. This hybrid is identical to the ideal-world protocol.

We now prove indistinguishability between each successive pair of hybrids.

Claim 2. *Assuming security of the TFHE scheme, the output of the simulator in H_0 is indistinguishable from the output of the simulator in H_1.*

Proof. This following directly from the security of the TFHE scheme, and from the random oracle assumption, that random oracle output is indistinguishable from uniform random string. □

Claim 3. *Assuming security of the short output compiler, the output of the simulator in H_1 is indistinguishable from the output of the simulator in H_2.*

Proof. This following directly from the security of short output compiler, and the fact that one-time pad distribution ($\mathsf{Combine}_\lambda$ output in H_1) is identical to uniform ($\mathsf{Combine}_\lambda$ output in H_2). □

Claim 4. *Assuming security of the SSB hash function, the puncturable PRF, and the iO scheme, the output of the simulator in H_2 is computationally indistinguishable from the output of the simulator in H_3.*

Proof. We prove indistinguishability between H_2 and H_3 via a sequence of two subhybrids $H_{2,j}$ for $j \in [M]$, where $H_{2,0} = H_1$ and $H_{2,M} = H_3$. In $H_{2,j}$, the simulator sets the output of the simulated $\mathsf{GenerateCircuit}$ protocol to be the circuit $\tilde{C}' = iO(\tilde{C}^j_{h,z_y,z_r,k,sk})$, where $\tilde{C}^j_{h,z_y,z_r,k,sk}$ is defined below. We show indistinguishability between each successive pair $H_{2,j-1}$ and $H_{2,j}$ by the following sequence of hybrids:

- **Hybrid $H_{2,j-1,1}$:** In this hybrid, the simulator behaves the same as in $H_{2,j-1}$ except that it sets the $h \leftarrow \mathsf{SSB.Setup}$, where the binding index $i^* = j$.
- **Hybrid $H_{2,j-1,2}$:** In this hybrid, the simulator behaves the same as in $H_{2,j-1,1}$, except that it uses an obfuscation of the circuit $\tilde{C}^{j,2}_{h,z_y,z_r,k',sk,y_j}$ in the output of $\mathsf{GenerateCircuit}$, with a punctured PRF key k' punctured at j, where $y_j = \perp$ if j is honest party.
- **Hybrid $H_{2,j-1,3}$:** In this hybrid, the simulator behaves the same as in $H_{2,j-1,2}$, except that from this point the simulator will answer to the random oracle queries as follows:

$$\mathcal{O}(x) = \begin{cases} PRF_k(i) \oplus y_i \oplus pad_i & x \notin cache \wedge x \in \{r_{seed}\|i\}_{i \in \mathcal{C} \wedge i \leq j} \\ cache(x) & x \in cache \wedge x \notin \{r_{seed}\|i\}_{i \in \mathcal{C} \wedge i \leq j} \\ \perp \text{ (and abort)} & x \in cache \wedge x \in \{r_{seed}\|i\}_{i \in \mathcal{C} \wedge i \leq j} \\ \text{uniform random string} & otherwise \end{cases}$$

- **Hybrid $H_{2,j-1,4}$:** In this hybrid, the simulator behaves the same as in $H_{2,j-1,3}$, except that it uses an obfuscation of the circuit $\tilde{C}^j_{h,z_y,z_r,k',sk}$ when computing the output of the simulated $\mathsf{GenerateCircuit}$ protocol. This hybrid is identical to $H_{2,j}$.

Circuit $\tilde{C}^j_{h,z_y,z_r,k,sk}(i, ct, r, \pi_y, \pi_r)$:

---

1. If $\mathsf{SSB.Verify}(h, z_y, i, ct, \pi_y) = 1$, and $\mathsf{SSB.Verify}(h, z_r, i, r, \pi_r) = 1$:
   (a) If $i \leq j$, output $r \oplus PRF_k(i)$.
   (b) Otherwise, output $\mathsf{TFHE.Dec}_{sk}(ct)$.
2. Otherwise, output $\perp$.

Circuit $\tilde{C}^{j,2}_{h,z_y,z_r,k',sk,out}(i, ct, r, \pi_y, \pi_r)$:

---

1. If $\mathsf{SSB.Verify}(h, z_y, i, ct, \pi_y) = 1$, and $\mathsf{SSB.Verify}(h, z_r, i, r, \pi_r) = 1$:

(a) If $i < j$, output $r \oplus PRF'_k(i)$.

(b) If $i = j$, output out.

(c) Otherwise, output $\mathsf{TFHE.Dec}_{sk}(ct)$.

2. Otherwise, output $\perp$.

We prove indistinguishability of each successive pair of these hybrids below. □

Claim 5. *Assuming the index hiding property of the SSB hash function, the output of the simulator in $H_{2,j-1}$ is indistinguishable from the output of the simulator in $H_{2,j-1,1}$.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between $H_{2,j-1}$ and $H_{2,j-1,1}$ with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against the index hiding property of the SSB hash function.

$\mathcal{A}'$ interacts with a challenger which supplies $\mathcal{A}'$ with an SSB hash key with the statistical binding bit i^* set to either $j-1$ or j. $\mathcal{A}'$ performs the same steps as the simulator in $H_{2,j-1}$, except that it queries the challenger for the SSB hash key h. It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

If the SSB index hiding challenger sends an SSB hash key with the statistical binding index i^* set to $j-1$, then the view of $\mathcal{A}$ is identical to $H_{2,j-1}$. If the challenger sends an SSB hash key with i^* set to j, the view of $\mathcal{A}$ is identical to $H_{2,j-1,1}$. This means that $\mathcal{A}'$ is a successful efficient adversary against the index hiding property of the SSB hash function. □

Claim 6. *Assuming security of the iO scheme and statistical binding of the SSB hash function, the output of the simulator in $H_{2,j-1,1}$ is indistinguishable from the output of the simulator in $H_{2,j-1,2}$.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between $H_{2,j-1,1}$ and $H_{2,j-1,2}$ with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against the security of the iO scheme.

$\mathcal{A}'$ interacts with a challenger, which receives two circuits from $\mathcal{A}'$ and sends the obfuscation of one of the two circuits. $\mathcal{A}'$ performs the same steps as the simulator in $H_{2,j-1,1}$, except that instead of computing the simulated output for $\mathsf{GenerateCircuit}$ directly, it sends the two circuits $C_0 = \tilde{C}^{j-1}_{h,z_y,z_r,k',sk}$ and $C_1 = \tilde{C}^{j,2}_{h,z_y,z_r,k',sk,y_j}$ to the challenger and receives back an obfuscation of one of them, which it uses as the simulated output of the $\mathsf{SSB.Setup}$. It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

If the iO challenger sends an obfuscation of C_0, then the view of $\mathcal{A}$ is identical to $H_{2,j-1,1}$. If the challenger sends obfuscation of C_1, the view of $\mathcal{A}$ is identical to $H_{2,j-1,2}$. This means that $\mathcal{A}'$ is a successful efficient adversary against the iO challenger, provided that C_0 and C_1 are functionally equivalent. Note that the only input where the behavior of C_1 could differ from the behavior of C_0 is on inputs (j, ct, r, π_y, π_r) , where $ct \neq ct_{o,j}$, the committed output ciphertext of party j, and $\mathsf{SSB.Verify}(h, z_y, j, ct, \pi_y) = 1$. But by the statistical binding property, this is impossible. □

Claim 7. *Assuming PRF assumption hold, and security of the puncturable PRF, the output of the simulator in $H_{2,j-1,2}$ is indistinguishable from the output of the simulator in $H_{2,j-1,3}$.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between $H_{2,j-1,2}$ and $H_{2,j-1,3}$ with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against security of the puncturable PRF.

$\mathcal{A}'$ interacts with a challenger which supplies $\mathcal{A}'$ with a PRF key k' punctured at position j, and either string α, the evaluation $PRF_k(j)$ with the unpunctured key k or a uniform random string of the same length. if $\mathcal{A}$ abort for such a α, $\mathcal{A}'$ also abort, if not $\mathcal{A}'$ performs the same steps as the simulator in $H_{2,j-1}$, except that it uses the challenger's key k' when computing the circuit $\tilde{C}^{j,2}_{h,sk,k',y,z_j}$, and program the random oracle as follows:

$$\mathcal{O}(x) = \begin{cases} \alpha \oplus y_i \oplus pad_i & x \notin cache \wedge x \in \{r_{seed} \| i\}_{i \in \mathcal{C} \wedge i = j} \\ PRF_k(i) \oplus y_i \oplus pad_i & x \notin cache \wedge x \in \{r_{seed} \| i\}_{i \in \mathcal{C} \wedge i < j} \\ cache(x) & x \in cache \wedge x \notin \{r_{seed} \| i\}_{i \in \mathcal{C} \wedge i \leq j} \\ \perp \text{ (and abort)} & x \in cache \wedge x \in \{r_{seed} \| i\}_{i \in \mathcal{C} \wedge i \leq j} \\ \text{uniform random string} & otherwise \end{cases}$$

It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

From the random oracle assumption and the fact that r_{seed} (the simulated output of $\mathsf{Combine}_\lambda$) is a uniform random string, $\mathcal{O}(r \| j)$ in $H_{2,j-1,2}$ is also uniform random string. So if the challenger sends a uniform random α, the view of $\mathcal{A}$ is identical to $H_{2,j-1,1,2}$. If the challenger sends $\alpha = PRF_k(j)$, then the view of $\mathcal{A}$ is identical to $H_{2,j-1,3}$. Note that if $\mathcal{A}$ will guess r_{seed} he can force $\mathcal{A}'$ to abort, but since the probability that $\mathcal{A}$ will guess $r_{seed} \in \{0,1\}^\lambda$ is $\mathsf{negl}(\lambda)$, this means that $\mathcal{A}'$ is a successful efficient adversary against security of the puncturable PRF. □

Claim 8. *Assuming security of the iO scheme and the statistical binding of the SSB hash function, the output of the simulator in $H_{2,j-1,3}$ is indistinguishable from the output of the simulator in $H_{2,j-1,4}$.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between $H_{2,j-1,3}$ and $H_{2,j-1,4}$ with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against the security of the iO scheme.

$\mathcal{A}'$ interacts with a challenger, which receives two circuits from $\mathcal{A}'$ and sends the obfuscation of one of the two circuits. $\mathcal{A}'$ performs the same steps as the simulator in $H_{2,j-1,3}$, except that instead of computing the simulated output for $\mathsf{GenerateCircuit}$ directly, it sends the two circuits $C_0 = \tilde{C}^{j,2}_{h,z_y,z_r,k',sk,y_j}$ and $C_1 = \tilde{C}^{j}_{h,z_y,z_r,k,sk}$ to the challenger and receives back an obfuscation of one of them, which it uses as the simulated output of the $\mathsf{SSB.Setup}$ protocol. It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

If the iO challenger sends an obfuscation of C_0, then the view of $\mathcal{A}$ is identical to $H_{2,j-1,3}$. If the challenger sends obfuscation of C_1, the view of $\mathcal{A}$ is identical to $H_{2,j-1,4}$. This means that $\mathcal{A}'$ is a successful efficient adversary against the iO challenger, provided that C_0 and C_1 are functionally equivalent. Note that the only input where the behavior of C_1 could differ from the behavior of C_0 is on inputs (j, ct, r, π_y, π_r) , where $r \neq r_{o,j}$, the committed randomness of party j, and $\mathsf{SSB.Verify}(h, z_r, j, r, \pi_r) = 1$. But by the statistical binding property, this is impossible. □

Claim 9. *Assuming semantic security of the TFHE scheme, the output of the simulator in H_3 is computationally indistinguishable from the output of the simulator in H_4.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between H_3 and H_4 with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against semantic security of the TFHE scheme.

$\mathcal{A}'$ performs the same steps as the simulator in H_2, except that instead of encrypting the honest parties' MPC inputs and randomness directly it sends the plaintexts $\{x_i\}_{i \notin \mathcal{C}}$ and $\{pad_i\}_{i \notin \mathcal{C}}$ to the challenger, and uses the ciphertexts received from the challenger as $\{ct_{\mathsf{st}_{i,0}}\}_{i \notin \mathcal{C}}$ and $\{ct_{pad_i}\}_{i \notin \mathcal{C}}$. It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

If the TFHE circular semantic security challenger sends ciphertexts with the true values, then the view of $\mathcal{A}$ is identical to H_3. If the challenger sends encryptions of 0 then the view of $\mathcal{A}$ is identical to H_4. This means that $\mathcal{A}'$ is a successful efficient adversary against the TFHE semantic security. □

8 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 8.1 (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 8.1 together with our short output semi-malicious compiler from Theorem A.1, we obtain a maliciously secure compiler for short output deterministic MPC protocols. Combining Theorem 8.1 together with our long output semi-malicious compiler from Theorem 7.1 we obtain a maliciously secure compiler for arbitrary deterministic MPC protocols. Full details are given in Section 8.5.

The rest of the section is organized as follows. In Section 8.1, we introduce the cryptographic primitives required for the protocol, introduce notation and make some simplifying assumptions. In Section 8.2, we define several subprotocols which we will use. In Section 8.3, we give the formal description of the compiler and analyze its efficiency, and finally, in Section 8.4, we prove that the compiled protocol satisfies the security properties specified in Theorem 8.1.

8.1 Ingredients, Assumptions and Notation

Ingredients. In our protocol we use several generic building blocks:

1. An identity-based simulation-extractable zkSNARK (idse-zkSNARK). By Theorem 3.3, it is sufficient to assume the existence of SNARKs for NP and one-way functions.

2. A threshold signature scheme with distributed reconstruction (Section 3.6). Such a scheme is known from LWE by adapting the construction of Boneh et al. [BGG+18]. Although [BGG+18] does not explicitly claim the distributed reconstruction property, it follows directly from the fact that their reconstruction procedure is essentially linear.

3. A collision resistant hash function family H. We assume there is an algorithm $\mathsf{H.Setup}(1^\lambda, 1^\gamma)$, which takes a security parameter λ and a compression parameter γ, and efficiently samples a hash function $h : \{0,1\}^{\lambda\gamma} \to \{0,1\}^\lambda$. Such a family is known from LWE. We also assume that size of the description of h and the space needed to evaluate h are both bounded by $\mathsf{poly}(\lambda) \cdot \gamma$.

4. A perfectly-binding non-interactive commitment scheme. Since any PKE scheme can be used as a perfectly binding commitment, without loss of generality we assume this exists from LWE.

5. A pseudo-random function family. Such a family is known from LWE.

Together, we obtain Theorem 8.1 from zkSNARKs and LWE, as stated.

Assumptions and notation. Throughout, if ct is a valid ciphertext for message m, then we assume $ct[\lambda \cdot (i-1) : \lambda \cdot i]$ is a valid ciphertext for the i-th bit of m (in particular, we assume that the blowup is λ bits, for simplicity). We now discuss some assumptions about the underlying P2P semi-malicious secure protocol, and we also fix some notation which will be useful when describing the compiler.

First, we assume the underlying P2P semi-malicious secure protocol operates in the PKI model. (It is straightforward to adapt our compiler to a semi-malicious protocol defined in the CRS model or the plain model as well.) For simplicity, we also assume the protocol operates with point-to-point authenticated (but not private) channels. In other words, an adversary can see all messages sent by honest parties, even if the recipient is also honest.

We assume this protocol's behavior is given in the form of a family of circuits $\{\mathsf{NextSt}_{i,r}\}_{i \in [M], r \in [R]}$, where $\mathsf{NextSt}_{i,r}$ denotes the behavior of party P_i in round r. We also assume $\mathsf{NextSt}_{i,r}$ has input and output which is given in a particular form which will simplify the description of the malicious-secure compiler. Recall from the technical overview that the high-level strategy of the compiler is that for each round r of the underlying protocol, the parties should collectively compute a Merkle root τ_r, where τ_r commits to a concatenation of all parties' states with respect to this underlying protocol at the end of round r. After obtaining τ_r, each party P_i must prove that its round-r behavior represented by τ_r was honest with respect to the previous round represented by τ_{r-1}. More specifically, P_i must show that it computed its round-r state honestly with respect to its round-$(r-1)$ state, and that it sent honest messages to all round-r recipients specified by the

$\mathsf{OutgoingMessageLocs}(i, r)$:	takes a party index i and a round r, and outputs an indexed family $\{(j, s_j, e_j)\}_j$, where each tuple (j, s_j, e_j) indicates that the sub-string of party P_i's state starting at s_j and ending at e_j should be sent to party P_j as a message.
$\mathsf{IncomingMessageLoc}(i, j, r)$:	takes a sending party index i, a receiving party index j, and a round r, and outputs a starting and ending location (s, e) in P_j's state where P_j will store the message from P_i at the end of round r.
$\mathsf{IncomingMessageGlobalLoc}(\lambda, i, j, r)$:	takes a security parameter λ, a sending party index i, a receiving party index j, and a round r, and outputs a starting and ending location (s, e) in the string committed to by τ_r, assuming the commitment scheme used has blowup λ, where P_j will store the message from P_i at the end of round r.
$\mathsf{StateLoc}(\lambda, i)$:	Takes security parameter λ and party index i and, assuming the commitment scheme $\mathsf{C.Commit}$ used has blowup λ, returns a location l where P_i's state will be stored in the Merkle tree.

Table 1: Helper Functions

protocol. Thus there must be a way, given τ_r, to prove relationships between messages sent by some party P_i and messages received by some other party P_j. The form which will be input and output by $\mathsf{NextSt}_{i,r}$ will facilitate this.

We describe this form now. $\mathsf{NextSt}_{i,r}$ takes input of the form $(\mathsf{smpk}, \mathsf{smsk}_i, \mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1}, \mathsf{msg}^{\mathsf{out}}_{i,r-1})$, where $(\mathsf{smpk}, \mathsf{smsk}_i)$ are the setup parameters, $\mathsf{st}_{i,r-1}$ is P_i's private state at the end of round $r-1$ and $\mathsf{msg}^{\mathsf{in}}_{i,r-1}$ contains all messages received by P_i during round $r-1$. Similarly, $\mathsf{NextSt}_{i,r}$ outputs a tuple of the form $(\mathsf{st}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r})$, where $\mathsf{st}_{i,r}$ is P_i's private state at the end of round r and $\mathsf{msg}^{\mathsf{out}}_{i,r}$ contains all messages sent by P_i during round r. The parties will compute τ_r as a Merkle root of the concatenation $\mathsf{C.Commit}(\mathsf{st}_{1,r})||\mathsf{msg}^{\mathsf{in}}_{1,r}||\mathsf{msg}^{\mathsf{out}}_{1,r}||\ldots||\mathsf{C.Commit}(\mathsf{st}_{M,r})||\mathsf{msg}^{\mathsf{in}}_{M,r}||\mathsf{msg}^{\mathsf{out}}_{M,r}$ of the outputs of $\mathsf{NextSt}_{i,r}$ for all parties, where each private state $\mathsf{st}_{i,r}$ is hidden using a noninteractive commitment.

Since the incoming and outgoing messages are sent in the clear, this enables each party P_i to prove that it sent honest messages to each recipient party P_j: it can simply prove that $(\mathsf{st}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r})$ was computed honestly, and as long as it has opening to each $\mathsf{msg}^{\mathsf{in}}_{j,r}$ at the appropriate location, it can simply prove that its outgoing message to P_j recorded in $\mathsf{msg}^{\mathsf{out}}_{i,r}$ is equal to the incoming message recorded in $\mathsf{msg}^{\mathsf{in}}_{j,r}$. We define some helper functions to help with this in Table 1

$\mathsf{Sim.Setup}_1(1^\lambda, 1^M) \to (\mathsf{smpk}, \mathsf{st}_{sim})$:	generates the simulated public key smpk and a private state st_{sim}.
$\mathsf{Sim.Setup}_2(\mathsf{st}_{sim}, \mathcal{C}) \to (\{\mathsf{smsk}_i\}_{i\in\mathcal{C}}, \mathsf{st}'_{sim})$:	takes the set of corrupted parties and the state from the previous step, and returns the set of simulated secret keys smsk_i along with an updated private state st_{sim}.
$\mathsf{Sim.Round}_1(\mathsf{st}_{sim}) \to (\{\mathsf{msg}^{\mathsf{out}}_{i,1}\}_{i\in[M]\setminus\mathcal{C}}, \mathsf{st}'_{sim})$:	For round 1, the simulator takes as input its private state st_{sim} and generates first-round messages on behalf of all honest parties along with an updated simulator's state st'_{sim}.
$\mathsf{Sim.Round}_r(\{(\mathsf{msg}^{\mathsf{out}}_{i,r-1}, x_i, r_i)\}_{i\in\mathcal{C}}, \mathsf{st}_{sim}) \to (\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}, \mathsf{st}'_{sim})$:	In all other rounds, the simulator takes as input the outgoing messages $\mathsf{msg}_{i,r-1}$ for each corrupted parties P_i, the explanation $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ for all corrupted parties' behavior up to this point, along with the simulator's private state st_{sim}, and generates first-round messages on behalf of all honest parties along with an updated simulator's state st'_{sim}, possibly querying the ideal functionality.

Table 2: Notation for P2P semi-malicious simulator

As the underlying MPC protocol is P2P semi-maliciously secure, it has a corresponding simulator which will be needed when proving security of the malicious compiler. We fix the notation for the simulator as follows, summarized in Table 2.

The setup phase is broken into two steps, $\mathsf{Sim.Setup}_1$ and $\mathsf{Sim.Setup}_2$. This is because we want the adversary to be able to choose the set $\mathcal{C}$ of corrupted parties after seeing the public setup parameters. So $\mathsf{Sim.Setup}_1$ receives as input a security parameter 1^λ and the number of parties 1^M and outputs a simulated public key smpk along with a simulator's private state. $\mathsf{Sim.Setup}_2$

takes this state along with the corrupted set $\mathcal{C}$, and returns an updated private state along with the simulated private setup parameters for the corrupted parties. For each round r, $\mathsf{Sim.Round}_r$ outputs the set $\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}$ of simulated outgoing messages for the honest parties for this round, along with the simulator's updated private state. During all rounds except for the first, $\mathsf{Sim.Round}_r$ takes the set $\{\mathsf{msg}^{\mathsf{out}}_{i,r-1}\}_{i\in\mathcal{C}}$ of outgoing messages for the corrupted parties from the previous round $r-1$. Since we are in the P2P semi-malicious setting, the simulator also takes as input a set $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ of inputs and randomness for all corrupted parties which completely explains the adversary's behavior up to this point. Note that we want to handle rushing adversaries, which generate their round-r messages after seeing the honest parties' messages, so the simulator does not see any round-r messages from corrupted parties when generating the honest parties' messages for round r. For this reason, during the first round, $\mathsf{Sim.Round}_1$ takes only its private state, and nothing else.

8.2 The Subprotocols

We define 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 6.

8.2.1 The CalcMerkleTree Subprotocol

First, we present 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 $\gamma = \lambda$. Each party is assigned to a set of nodes in the tree. The 0-th level of the tree consists of the leaves, one for each party, and each party supplies a string x_i corresponding to that leaf. The t-th level consists of the Merkle root, which is assigned to P_0. The job of P_0 will be to receive the level-$t-1$ labels and to apply the hash function to compute the final root label, and then to distribute the root and the level-$t-1$ openings up the tree. The nodes on the interior levels of the tree have a similar job: receive labels from their children, compute the label corresponding to this node by applying the hash function, and send this label to the node's parent and the openings back to the children.

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.

Protocol 3 $\mathsf{CalcMerkleTree}_h(\{x_i\}_{i\in[M]})$:

Input: Each party P_i has a hash function h and an input x_i, where $|x_i|\leq S$. In this protocol we will number the parties starting at 0 (so the first party will be P_0).

Parameters: Let $\lambda \leq S$ and $t = \lceil\log_\gamma M\rceil$. Assume h is a collision-resistant hash function which has been initialized with security parameter λ and compression parameter $\gamma = \lambda$.

Before starting: Each party P_i first computes a Merkle root $y_{i,0}$ for a $\log|x_i|$-depth, fanin-2 Merkle tree of its input x_i.

When carrying out the protocol, we will divide the parties into subsets. Let $\mathcal{S}_r = \{P_i \mid i \equiv 0 \pmod{\gamma^r}\}$ (and let $\mathcal{S}_0 = \{P_i\}_{i\in[M]}$), let the set of children for i in $\mathcal{S}_r$ be $\mathcal{D}_{i,r} = \{P_j \mid j \equiv 0 \pmod{\gamma^{r-1}}$ and $i \le j \le i + \gamma^r\}$, and let the parent of i in $\mathcal{S}_r$ be $q_{i,r} = \gamma^r \lfloor i/\gamma^r \rfloor$.
For $k = 1, \ldots, \lceil \log_\gamma M \rceil + 1$, do the following:

Round k: In this round, the parties in the sets $\mathcal{S}_{k-t}$, $t = 1, 3, \ldots, 2\lceil k/2 \rceil - 1$ will participate.

- Each party P_i in $\mathcal{S}_{k-1}$ does the following:
  1. If $k - 1 > 0$, receive $y_{j,k-2}$ from each $P_j \in \mathcal{D}_{i,k-1}$
  2. If $k - 1 > 0$, calculate $y_{i,k-1} \leftarrow h(\{y_{j,k-2}\}_{P_j \in \mathcal{D}_{i,k-1}})$, and send $(y_{i,k-1}, \{y_{j,k-2}\}_{j\in\mathcal{D}_{i,k-1}})$ to all parties P_j, $j \in \mathcal{D}_{i,k-1}$.
  3. Send $y_{i,k-1}$ to P_{q_i}
- Each party P_j in $\mathcal{S}_r$ for $r = 0, \ldots, k-2$ does the following:
  1. Check if received $\theta_{j,r'} = (y_{i,r'}, \{y_{j',r'-1}\}_{j'\in\mathcal{D}_{i,r'}})$ from P_{q_j}
  2. If so, append $\theta_{j,r'}$ to θ_j.
  3. If $r > 0$, send $\theta_{j,r'}$ to all $P_{j''} \in \mathcal{D}_{j,r}$.

For $k' = \lceil \log_\gamma M \rceil + 2, \ldots, 2\lceil \log_\gamma M \rceil + 1$:

Round k': Each party P_j in $\mathcal{S}_r$ for $r = 0, \ldots, \lceil \log_\gamma M \rceil$ does the following:

1. Check if received $\theta_{j,r'} = (y_{i,r'}, \{y_{j',r'-1}\}_{j'\in\mathcal{D}_{i,r'}})$ from P_{q_j}
2. If so, append $\theta_{j,r'}$ to θ_j.
3. If $r > 0$, send $\theta_{j,r'}$ to all $P_{j''} \in \mathcal{D}_{j,r}$.

Output: The protocol stops after $2\lceil \log_\gamma M \rceil$ rounds, and every party P_i knows the Merkle tree root y and the opening θ_i of x_i. Moreover, θ_i can be divided into openings $\theta_{i,j}$ for each size-λ chunk of x_i, where each opening $\theta_{i,j}$ has size $\log|x_i|\cdot 2 + \mathsf{poly}(\lambda)$.

We briefly discuss correctness and efficiency of $\mathsf{CalcMerkleTree}$.

Correctness: We say that correctness holds for $\mathsf{CalcMerkleTree}$ if the end of an honest execution of the protocol each party P_i knows the root label y of the tree along with an opening θ_i corresponding to P_i's input x_i. Note that each P_i receives messages for each ancestor of the leaf node $y_{i,0}$ corresponding to P_i's input, and each message is of the form $(y_{i',r}, \{y_{j,r-1}\}_{j\in D_{i',r}})$, containing the ancestor's label and the ancestor's direct children's labels. This is sufficient to learn the root label z of the tree as well as all node labels along the path to $y_{i,0}$ along with the labels of the siblings of all nodes along this path. From this, P_i can construct θ_i.

Efficiency: The $\mathsf{CalcMerkleTree}$ protocol has the property that each party P_i takes local space bounded by $S + \mathsf{poly}(\lambda)$, assuming that for all $i, |x_i| \le S$ and $h \leftarrow \mathsf{H.Setup}(1^\lambda)$. A straightforward analysis shows that the noninteractive phase and each round take space bounded by $S \cdot \mathsf{poly}(\lambda)$, and because $M \in \mathsf{poly}(\lambda)$ there are a constant number $2\lceil \log_\lambda M \rceil + 1 = O(1)$ of rounds. These two facts together imply the space bound.

8.2.2 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 Section 3.6. The distributed reconstruction property is used to achieve the required space efficiency properties.

Protocol 4 $\mathsf{Agree}_\gamma(\{x_i\}_{i\in[M]}, \mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]})$:

Input: Each party P_i has a string x_i of size at most λ, the public verification key vk, and a secret key ssk_i, where $(\mathsf{vk}, \{\mathsf{ssk}_i\}) \leftarrow \mathsf{Sig.Setup}(1^\lambda, 1^M)$ were generated using the TSDR setup algorithm.

1. Each party P_i computes the signature $\sigma_i \leftarrow \mathsf{Sig.Sign}_{\mathsf{ssk}_i}(x_i)$ of the string x_i with its secret key ssk_i.
2. The parties run $\mathsf{Combine}_\gamma(\mathsf{Sig.Combine}, \{\sigma_i\})$ so that P_1 has the combined signature σ.
3. The parties run $\mathsf{Distribute}_\gamma(\sigma)$.
4. Each party P_i runs the verification algorithm $\mathsf{Sig.Verify}_{\mathsf{vk}}(x_i, \sigma)$ on the combined signature and aborts if the verification fails.

Output: Each party P_i outputs x_i.

Correctness: We say that Agree satisfies correctness if for any honest execution of the subprotocol where $x_i = x_j$ for all $i, j \in [M]$, the protocol does not abort and every party P_i outputs x_i at the end. Correctness follows directly from the correctness of the TSDR scheme.

Efficiency: The Agree protocol has the property that each party P_i takes local space bounded by $S \cdot \mathsf{poly}(\lambda)$, assuming that for all i $|x_i| \leq \lambda$. This follows directly from the efficiency properties of the TSDR scheme.

Security: We define the security properties needed for Agree in terms of a game AgreeSecurity, and prove that Agree satisfies these requirements.

Lemma 8.1. *Let $\mathcal{A}$ be any PPT adversary. Then assuming unforgeability of the threshold signature scheme, for all R,* $\Pr\left[\mathsf{AgreeSecurity}_{\mathcal{A}}(1^\lambda, R) < \mathsf{negl}(\lambda)\right]$, *where* AgreeSecurity *is nhe game defined below.*

Game 5 $\mathsf{AgreeSecurity}_{\mathcal{A}}(\lambda, R)$:

---

1. Initialize $\mathcal{A}$ with security parameter 1^λ.
2. $(\mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]}) \leftarrow \mathsf{Sig.Setup}(1^\lambda, M)$
3. Send vk to $\mathcal{A}$ and receive $\mathcal{C} \subsetneq [M]$ from $\mathcal{A}$.

4. Send $\{\mathsf{ssk}_i\}_{i\in\mathcal{C}}$ to $\mathcal{A}$.

5. For $r \in \{0, \dots, R\}$:

   (a) Receive $\{x_i\}_{i\in[M]\setminus\mathcal{C}}$ from $\mathcal{A}$.

   (b) Execute an instance of Agree, acting on behalf of each party P_i, $i \in [M] \setminus \mathcal{C}$ using $((r, x_i), \mathsf{vk}, \mathsf{ssk}_i)$ as its input, and interacting with $\mathcal{A}$ which provides the behavior of the corrupted parties.

   (c) If Agree does not abort and there exists $i, j \in [M] \setminus \mathcal{C}$ such that P_i's output x_i is not equal to P_j's output x_j, then halt and output 1.

6. Output 0.

Proof. Assume there is a PPT $\mathcal{A}$ which causes $\mathsf{AgreeSecurity}_{\mathcal{A}}(\lambda, R)$ to output 1 with non-negligible probability. We build a reduction $\mathcal{A}'$ to the Unforgeability game from Definition 13.

$\mathcal{A}'$ is an adversary for Unforgeability which has $\mathcal{A}$ hardcoded, and enacts AgreeSecurity with $\mathcal{A}$, except for the following differences:

- Instead of generating the setup itself, $\mathcal{A}'$ queries the Unforgeability challenger to receive vk which it forwards to $\mathcal{A}$, and passes the choice of $\mathcal{C}$ made by $\mathcal{A}$ to the challenger to receive $\{\mathsf{ssk}_i\}_{i\in\mathcal{C}}$.

- During the first step of the Agree protocol for each round r, instead of generating the honest parties' signatures directly, $\mathcal{A}'$ sends $(\{i\}, (r, x_i))$ to the challenger for each honest party P_i, and receives back a signature which it uses as P_i's signature.

- During each round r, if Agree does not abort and there exists $i, j \in [M] \setminus \mathcal{C}$ such that P_i's output x_i is not equal to P_j's output x_j, then $\mathcal{A}'$ outputs the signature σ received by P_i at the end of Agree.

Whenever it is the case that Agree does not abort and there exists $i, j \in [M] \setminus \mathcal{C}$ such that P_i's output x_i is not equal to P_j's output x_j, it follows that $\mathcal{A}'$ did not query the challenger for $(\{j\}, (r, x_i))$ during round r. This means that $((r, x_i), [M])$ is not in the challenger's set $\mathcal{S}$. On the other hand, since all parties accepted, in particular this means that for the signature σ received by P_i during Agree, it holds that $\mathsf{Sig.Verify}_{\mathsf{vk}}((r, x_i), \sigma)$. Thus $\mathcal{A}'$ wins the unforgeability game in this case. Since this happens whenever $\mathcal{A}$ wins AgreeSecurity, which happens with non-negligible probability, $\mathcal{A}'$ wins Unforgeability with non-negligible probability. □

8.2.3 The SNARK statements and the RecCompAndVerify Subprotocol

The last subprotocol, RecCompAndVerify, deals with recursive composition and verification of the zk-SNARKs 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.

Before defining the subprotocol, we first define formally the statements used when computing zk-SNARKs. At a high level, the statement $\Phi((i, r, 0, \tau_{r-1}, \tau_r), w)$ 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. We define the statements by specifying algorithms which take the statement and the witness as inputs, and output 1 if the witness is valid for the statement.

Statement 1 $\Phi((i,r,0,\tau_{r-1},\tau_r),(c_{\mathsf{st}_{i,r-1}},\mathsf{msg}^{\mathsf{in}}_{i,r-1},\mathsf{msg}^{\mathsf{out}}_{i,r-1},\theta_{i,r-1},c_{\mathsf{st}_{i,r}},\mathsf{msg}^{\mathsf{in}}_{i,r},\mathsf{msg}^{\mathsf{out}}_{i,r},\theta_{i,r},k_i,c_{k_i},$ $\alpha_{k_i},\mathsf{st}_{i,r-1},\mathsf{st}_{i,r},\ \{(m_{j,r},\theta_{m_{j,r}})\}_j))$:

1. Verify that the Merkle openings to the commitment to the private state of P_i along with the messages sent and received during rounds $r-1$ and r are valid with respect to τ_{r-1} and τ_r, and verify that the opening for the commitments to the private states are valid:
   (a) Compute $l_i \leftarrow \mathsf{StateLoc}(\lambda, i)$, the location of P_i's encrypted state in the Merkle trees.
   (b) If $\theta_{i,r}$ does not open τ_r at location l_i to $c_{\mathsf{st}_{i,r}}||\mathsf{msg}^{\mathsf{in}}_{i,r}||\mathsf{msg}^{\mathsf{out}}_{i,r}||c_{k_i}$, then halt and output 0.
   (c) If $c_{k_i} \neq \mathsf{C.Commit}(k_i;\alpha_{k_i})$, then halt and output 0.
   (d) if $c_{\mathsf{st}_{i,r}} \neq \mathsf{C.Commit}(\mathsf{st}_{i,r};\mathsf{PRF}_{k_i}(r))$, then halt and output 0.
   (e) If $r=0$, then halt and output 1. Otherwise, continue.
   (f) If $\theta_{i,r-1}$ does not open τ_{r-1} at location l_i to $c_{\mathsf{st}_{i,r-1}}||\mathsf{msg}^{\mathsf{in}}_{i,r-1}||\mathsf{msg}^{\mathsf{out}}_{i,r-1}||c_{k_i}$, then halt and output 0.
   (g) if $c_{\mathsf{st}_{i,r-1}} \neq \mathsf{C.Commit}(\mathsf{st}_{i,r-1};\mathsf{PRF}_{k_i}(r))$, then halt and output 0.
2. Verify that computation and copying of messages was performed honestly:
   (a) If $\mathsf{NextSt}_{i,r}(\mathsf{st}_{i,r-1},\mathsf{msg}^{\mathsf{in}}_{i,r-1}) \neq (\mathsf{st}_{i,r},\mathsf{msg}^{\mathsf{out}}_{i,r})$, halt and output 0.
   (b) For each (j,s_j,e_j) in $\mathsf{OutgoingMessageLocs}(i,r)$:
      i. if $\theta_{m_{j,r}}$ does not open τ_r at location $\mathsf{IncomingMessageGlobalLoc}(\lambda,i,j)$ to $m_{j,r}$, halt and output 0.
      ii. $\mathsf{msg}^{\mathsf{out}}_{i,r}[s_j : e_j] \neq m_{j,r}$, halt and output 0.
3. Output 1.

Statement 2 $\Phi((i,r,k>0,\tau_{r-1},\tau_r),(\{\pi_{j,k-1}\}_{j\equiv 0 \pmod{\gamma^{k-1}},\lfloor j/\gamma^k\rfloor=i}))$:

1. If $i \not\equiv 0 \pmod{\gamma^k}$, output 0.
2. Verify that for each j such that $j \equiv 0 \pmod{\gamma^{k-1}}$, $\lfloor j/\gamma^k \rfloor = i$, $\Pi.\mathsf{V}(crs,\Phi(j,r,k-1,\tau_{r-1},\tau_r),\pi_{j,k-1}) = 1$; otherwise, output 0.

We now define the $\mathsf{RecCompAndVerify}$ protocol below. Note that each party P_i has a unique id (i,k) which it will use only during round k. We also assume that the SNARKs which are given as inputs to $\mathsf{RecCompAndVerify}$ are proven using ids $(i,0)$. This will be useful when proving security of the subprotocol, where we will want to extract from all SNARKs except for the ones generated by the honest parties at the top level.

Protocol 5 $\mathsf{RecCompAndVerify}(r,crs,\tau_{r-1},\tau_r,\{(\pi_{i,r})\})$:

Parameters: Let the fan-in γ be λ.

Start: Each node P_i sets $\pi_{i,r,0} = \pi_{i,r}$.

Round k: In this round, the parents are all P_i such that $i \equiv 0 \pmod{\gamma^k}$, and the children are all P_j such that $j \equiv 0 \pmod{\gamma^{k-1}}$ but $j \not\equiv 0 \pmod{\gamma^k}$. The round proceeds as follows:

1. Each child P_j sends $\pi_{j,r,k-1}$ its parent P_i.
2. P_i computes the zk-SNARK $\pi_{i,r,k} \leftarrow \Pi.\mathsf{P}(crs, \Phi(i, r, k, \tau_{r-1}, \tau_r, (i, k)), \{\pi_{j,r,k-1}\})$.

Proof Verification: After $t = \lceil \log_\gamma M \rceil$ rounds, P_1 has $\pi_{0,r,t} = \pi_r$ which proves that the round r transcript committed to by τ_r has been honestly computed with respect to the round $r-1$ transcript committed to by τ_{r-1}. The parties then do the following:

1. The parties run the subprotocol $\mathsf{Distribute}_\gamma(\pi_r)$ so that every party obtains π_r.
2. Each party P_i runs $\Pi.\mathsf{V}(crs, \Phi(0, r, t, \tau_{r-1}, \tau_r), \pi_r, 1)$. If verification fails, P_i aborts and stops responding.

Output: Each party P_i outputs π_r.

Correctness: We say that $\mathsf{RecCompAndVerify}$ satisfies correctness if for every r, valid idse-zkSNARK setup parameter crs, trees τ_{r-1}, τ_r, and proofs $\{\pi_{i,r}\}$ such that for all i $\Pi.\mathsf{V}(crs, \Pi(i, r, 0, \tau_{r-1}, \tau_r), \pi_{i,r}, i) = 1$, an honest execution of the protocol does not abort, and every party P_i outputs π_r such that $\Pi.\mathsf{V}(crs, \Pi(0, r, t, \tau_{r-1}, \tau_r), \pi_r, 1) = 1$. Correctness follows directly from correctness of the idse-zkSNARK protocol.

Efficiency: The $\mathsf{RecCompAndVerify}$ protocol has the property that each party P_i takes local space bounded by $S \cdot \mathsf{poly}(\lambda)$, assuming that for all i $|x_i| \leq \lambda$. This follows directly from the efficiency properties of the idse-zkSNARK scheme.

Security: We define the security properties needed for $\mathsf{RecCompAndVerify}$, in terms of a game $\mathsf{RCVSecurity}$. In this game, a nonuniform PPT adversary $\mathcal{A}$ invokes R sequential instances of $\mathsf{RecCompAndVerify}$. 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 $\mathsf{RecCompAndVerify}$ 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.

Lemma 8.2. *Let $\mathcal{A}$ be any PPT adversary. Then assuming unforgeability of the threshold signature scheme along with simulation-extractability of the idse-zkSNARK scheme, there is a PPT machine $\mathcal{E}_\mathcal{A}$ such that for all R, r_1 and r_2, it holds that*

$$\Pr\left[\mathsf{RCVSecurity}_{\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda, R, r_1, r_2)\right] < \mathsf{negl}(\lambda),$$

where $\mathsf{RCVSecurity}$ *is the game defined below.*

Game 6 $\mathsf{RCVSecurity}_{\mathcal{A},\mathcal{E}_\mathcal{A}}(1^\lambda, R, r_1, r_2)$:

1. Choose uniform random strings $\alpha_{snark}, \alpha_\mathcal{A}, \{\alpha_{i,r}\}_{i \in [M], r \in \{0,\ldots,R\}}$.

2. Generate a simulated idse-zkSNARK setup $(\widetilde{crs}, \mathsf{td}) \leftarrow \Pi.\mathsf{Sim}_1(1^\lambda, M \cdot (t+1); \alpha_{snark})$.
3. Initialize $\mathcal{A}$ with security parameter 1^λ, setup parameters $\widetilde{crs}$, and randomness $\alpha_{\mathcal{A}}$.
4. Receive the corrupted set $\mathcal{C}$ from $\mathcal{A}$.
5. For each $i \in [M] \setminus \mathcal{C}$, generate $\mathsf{td}_i \leftarrow \Pi.\mathsf{Sim}_2(\widetilde{crs}, \mathsf{td}, (i, 0))$.
6. Set $\tau_{-1} = \perp$.
7. For each $r \in \{0, \dots, R\}$:
   (a) Receive τ_r from $\mathcal{A}$.
   (b) For each $i \in [M] \setminus \mathcal{C}$, compute a simulated proof $\pi_{i,r} \leftarrow \Pi.\mathsf{Sim}_3(\widetilde{crs}, i, \mathsf{td}_i, \Phi(i, r, 0, \tau_{r-1}, \tau_r); \alpha_{i,r})$.
   (c) Execute an instance of $\mathsf{RecCompAndVerify}$, acting on behalf of each party P_i, $i \in [M] \setminus \mathcal{C}$ using $(r, \widetilde{crs}, \tau_{r-1}, \tau_r, \pi_{i,r})$ as its input, and interacting with $\mathcal{A}$ which provides the behavior of the corrupted parties. If any parties abort, halt and output 0.
   (d) If $r = r_1$ or $r = r_2$:
      i. Compute $\{w_i\}_{i \in \mathcal{C}} \leftarrow \mathcal{E}_{\mathcal{A}}(r_1, r, \widetilde{crs}, \alpha, \{\mathsf{td}_i\}_{i \in [M] \setminus \mathcal{C}}, \{\alpha_{i,r'}\}_{i \in [M] \setminus \mathcal{C}, r' \in \{0,\dots,r\}})$, where $\alpha = (\alpha_{\mathcal{A}}, \{\alpha_{i,r}\})$. If the extractor fails, output 1.
      ii. If there is an $i \in \mathcal{C}$ such that $\Phi((i, r, 0, \tau_{r-1}, \tau_r), (w_i)) = 0$, halt and output 1.
8. Output 0.

Proof. We divide the proof of this lemma into two parts. First, we define the extractor $\mathcal{E}_{\mathcal{A}}$. Next, we prove by induction that with respect to $\mathcal{E}_{\mathcal{A}}$, $\mathcal{A}$ wins $\mathsf{RCVSecurity}$ with at most negligible probability.

Defining the extractor: We define $\mathcal{E}_{\mathcal{A}}$ by first recursively defining a large family of extractors. Define $\mathcal{A}'$ as a machine that has $\mathcal{A}$ and r_1 hardcoded and takes $\widetilde{crs}$ and randomness $\alpha = (\alpha_{\mathcal{A}}, \{\alpha_{i,r}\}_{i \in [M], r \in \{0,\dots,R\}})$ as input, and plays $\mathsf{RCVSecurity}_{\mathcal{A},\_}(1^\lambda, R, r_1, r_2)$ with the following differences:

- $\mathcal{A}'$ receives a crs $\widetilde{crs}$ from the challenger and forwards it to $\mathcal{A}$.
- After $\mathcal{A}$ returns the corrupted set $\mathcal{C}$, $\mathcal{A}'$ sends $\{(i, r)\}_{i[M], r \in \{1,\dots,t\}} \cup \{(i, 0)\}_{i \in \mathcal{C}}$ to a challenger and receives back $\{\mathsf{td}_{(i,0)}\}_{i \in [M] \setminus \mathcal{C}}$. $\mathcal{A}'$ uses these trapdoors during each round in order to simulate the honest parties' proofs.
- If $r = r_1$, instead of executing step 7d above, $\mathcal{A}'$ halts and outputs π_r which was output by P_1 from the $\mathsf{RecCompAndVerify}$ subprotocol.

Now using $\mathcal{A}'$ and Definition 9, we can recursively define extractors $\mathsf{ExtPart}_{\mathcal{A}}([r_1], [j_1 \dots, j_k], \widetilde{crs}, \alpha, \{\mathsf{td}_i\})$ and $\mathsf{ExtPart}_{\mathcal{A}}([r_1, r_2], [j_1 \dots, j_k], \widetilde{crs}, \alpha, \{\mathsf{td}_i\})$ for each valid choice of $[j_1, \dots, j_k] \in \mathbb{Z}_\gamma^{\leq t}$, where $\mathsf{ExtPart}_{\mathcal{A}}([r_1], [j_1 \dots, j_k], \widetilde{crs}, \alpha, \{\mathsf{td}_i\})$ extracts the proof $\pi_{j_k, r_1, (t-k)}$ from $\mathsf{RecCompAndVerify}$ in round r_1 and $\mathsf{ExtPart}_{\mathcal{A}}([r_1, r_2], [j_1 \dots, j_k], \widetilde{crs}, \alpha, \{\mathsf{td}_i\})$ extracts the proof $\pi_{j_k, r_2, (t-k)}$ from $\mathsf{RecCompAndVerify}$ in round r_2.

We start with $\mathsf{ExtPart}_{\mathcal{A}}([r_1], [j_1 \dots, j_k], \widetilde{crs}, \alpha, \{\mathsf{td}_i\})$, which we define recursively in terms of $[j_1, \dots, j_k] \in \mathbb{Z}_\gamma^{\leq t}$:

- **Base case:** $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ runs the extractor $\mathcal{E}_{\mathcal{A}'}$ from Definition 9 on the same arguments, which outputs the set $\{\pi_{j,r,t-1}\}_{j\equiv 0 \pmod{\gamma^{t-1}}}$ of proofs from the second-to-last round of the protocol $\mathsf{RecCompAndVerify}$. $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ then then simply outputs $\pi_{j_1,r,t-1}$ from this set.

- **Recursive case:** Assuming $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1,\ldots,j_{k-1}],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ is defined, we recursively define $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ as follows. Since $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_{k-1}],\cdot,\cdot,\cdot)$ is a machine that takes a simulated crs $\widetilde{crs}$, some randomness, and the trapdoors for $i\notin\mathcal{C}$, it can also be used as an adversary in the extraction game of Definition 9. Thus we have an extractor $\mathcal{E}_{\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_{k-1}],\cdot,\cdot,\cdot)}$. $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ runs $\mathcal{E}_{\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_{k-1}],\cdot,\cdot,\cdot)}$ on the same arguments to obtain the set $\{\pi_{j,r,t-k}\}_{j\equiv 0 \pmod{\gamma^{t-k}},\lfloor j/\gamma^{t-k}\rfloor=j_{k-1}}$, and outputs $\pi_{j_k,r,t-k}$ from this set.

Now let $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ run $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1,\ldots,j_t],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ for every valid $[j_1,\ldots,j_t]$ in order to get $\{w_i\}_{i\in\mathcal{C}}$. Here "valid" means that $[j_1,\ldots,j_t]$ is a valid path in the tree of SNARKs defined by $\mathsf{RecCompAndVerify}$ such that $j_t\in\mathcal{C}$. $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ returns this set, or aborts if any of the extractors $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1,\ldots,j_t],\cdot,\cdot,\cdot)$ aborts.

Let $\mathcal{A}''$ be a machine which has $\mathcal{A}$, r_1, r_2 and $\mathsf{Ext}_{\mathcal{A}}([r_1],\cdot,\cdot,\cdot)$ hardcoded, and runs $\mathsf{RCVSecurity}_{\mathcal{A},\mathsf{Ext}_{\mathcal{A}}([r_1],\cdot,\cdot,\cdot)}(1^\lambda,R,r_1,r_2)$ with the following differences:

- $\mathcal{A}'$ receives a crs $\widetilde{crs}$ from the challenger and forwards it to $\mathcal{A}$.

- After $\mathcal{A}$ returns the corrupted set $\mathcal{C}$, $\mathcal{A}''$ sends $\mathcal{C}$ to a challenger and receives back $\{\mathsf{td}_i\}_{i\in[M]\setminus\mathcal{C}}$. $\mathcal{A}''$ uses these trapdoors during each round in order to simulate the honest parties' proofs.

- If $r=r_2$, instead of executing step 7d above, $\mathcal{A}''$ halts and outputs π_r which was output from the $\mathsf{RecCompAndVerify}$ subprotocol. (If all honest parties do not agree on π_r then $\mathcal{A}''$ aborts.)

We define $\mathsf{ExtPart}_{\mathcal{A}}([r_1,r_2],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ in terms of $\mathcal{A}''$ in exactly the same way as $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ was defined in terms of $\mathcal{A}'$, and we define $\mathsf{Ext}_{\mathcal{A}}([r_1,r_2],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ in terms of $\mathsf{ExtPart}_{\mathcal{A}}([r_1,r_2],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ in exactly the same way as $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ was defined in terms of $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$.

Having defined $\mathsf{Ext}_{\mathcal{A}}([r_1,r_2],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ and $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$, we let $\mathcal{E}_{\mathcal{A}}$ implement $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ when extracting during round r_1 and $\mathsf{Ext}_{\mathcal{A}}([r_1,r_2],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ when extracting during round r_2.

Proving negligible probability of winning: First we note that every extractor which was defined in the previous paragraphs runs in polynomial-time in λ. This is because the maximum number of compositions $2(t+1)$ of the extractor from Definition 9 is constant in λ.

Now it only remains to show that $\mathsf{RCVSecurity}_{\mathcal{A},\mathcal{E}_{\mathcal{A}}}(\lambda,R,r_1,r_2)$ outputs 1 with negligible probability in λ. Assume that during round r_1 of the $\mathsf{RCVSecurity}$ experiment, The $\mathsf{RecCompAndVerify}$ protocol outputs a π_{r_1} such that $\Pi.\mathsf{V}(\widetilde{crs},\Phi(0,r,t,\tau_{r_1-1},\tau_{r_1}),\pi_{r_1})=1$ with non-negligible probability. We prove by induction the following statement: for all valid $[j_1,\ldots,j_k]$ such that $k<t$, conditioned on $\mathsf{RecCompAndVerify}$ outputting a valid π_{r_1} as explained above, with overwhelming probability $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ outputs a $\pi_{j_k,r_1,t-k}$ such that $\Pi.\mathsf{V}(\widetilde{crs},\Phi(j_k,r_1,t-k,\tau_{r_1-1},\tau_{r_1}),\pi_{j_k,r_1,t-k},(j_k,t-k))=1$.

We first prove the base case. Assume that the base case does not hold. Then with non-negligible probability conditioned on $\mathsf{RecCompAndVerify}$ in round r_1 outputting a valid π_{r_1}, $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ fails to output a valid $\pi_{j_1,r_1,t-1}$ such that $\Pi.\mathsf{V}(\widetilde{crs},\Phi(j_1,r_1,t-1,\tau_{r_1-1},\tau_{r_1}),\pi_{j_1,r_1,t-1},(j_1,t-1))=1$. Note that $\mathcal{A}'$ outputs exactly the π_{r_1} from the

$\mathsf{RecCompAndVerify}$ protocol. Consider the game $\mathsf{SimExt}_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(\lambda)$. Since $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1],\cdot,\cdot,\cdot)$ fails to output a valid proof with non-negligible probability, and $\mathcal{A}'$ does not receive a trapdoor for id $(j_1, t-1)$, this means that $\mathsf{SimExt}_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(\lambda)$ outputs 1 with non-negligible probability, contradicting simulation-extractability of the idse-zkSNARK scheme.

We now prove the induction step. Assume that for some $k < t$, with non-negligible probability conditioned on $\mathsf{RecCompAndVerify}$ in round r_1 outputting a valid π_{r_1}, $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ fails to output a valid $\pi_{j_k,r_1,t-k}$ such that $\Pi.\mathsf{V}(\widetilde{crs},\Phi(j_k,r_1,t-k,\tau_{r_1-1},\tau_{r_1}),\pi_{j_k,r_1,t-k},(j_k,t-k)) = 1$. By the induction hypothesis, with overwhelming conditional probability $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1,\ldots,j_{k-1}],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ outputs a valid $\pi_{j_{k-1},r_1,t-(k-1)}$ such that $\Pi.\mathsf{V}(\widetilde{crs},\Phi(j_{k-1},r_1,t-(k-1),\tau_{r_1-1},\tau_{r_1}),\pi_{j_{k-1},r_1,t-(k-1)},(j_{k-1},t-(k-1))) = 1$. Letting $\mathcal{A}_{hyp} = \mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_{k-1}],\cdot,\cdot,\cdot))$, Consider the game $\mathsf{SimExt}_{\mathcal{A}_{hyp},\mathcal{E}_{\mathcal{A}_{hyp}}}(\lambda)$. Since $\mathsf{ExtPart}_{\mathcal{A}}([r_1],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ is defined in terms of $\mathcal{E}_{\mathcal{A}_{hyp}}$ and fails to output a valid proof with non-negligible probability, and $\mathcal{A}_{hyp}$ does not receive a trapdoor for id $(j_k, t-k)$, this means that $\mathsf{SimExt}_{\mathcal{A}_{hyp},\mathcal{E}_{\mathcal{A}_{hyp}}}(\lambda)$ outputs 1 with non-negligible probability, contradicting simulation-extractability of the idse-zkSNARK scheme.

By essentially the same argument, we can also show $\mathsf{Ext}_{\mathcal{A}}([r_1],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ outputs a set of valid witnesses with overwhelming probability conditioned on $\mathsf{RecCompAndVerify}$ outputting a valid proof in r_1. Then by applying the same argument again, we can show the same thing for all $\mathsf{ExtPart}_{\mathcal{A}}([r_1,r_2],[j_1\ldots,j_k],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ and $\mathsf{Ext}_{\mathcal{A}}([r_1,r_2],\widetilde{crs},\alpha,\{\mathsf{td}_i\})$ as well. Since $\mathcal{E}_{\mathcal{A}}$ is defined in terms of these two extractors, this completes the proof of the lemma.

□

8.3 The Compiler

We now give the formal description of the compiler.

Protocol 6 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 $\pi_{0,r} \leftarrow \Pi.\mathsf{P}(crs,\Phi(i,0,\perp,\tau_0),$
$(\perp,\perp,\perp,c_{\mathsf{st}_{i,0}},\perp,\perp,\theta_{i,0},k_i,c_{k_i},\alpha_{k_i},\perp,\mathsf{st}_{i,0},\perp),(i,0))$.

5. All parties run $\mathsf{RecCompAndVerify}(crs, \perp, \tau_0, \{\pi_{i,0}\}_{i\in[M]})$ to obtain and verify π_0, a SNARK for the statement $\Phi(0, 0, t, \perp, \tau_0)$. 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 $\widetilde{\Pi}$ is at most $S \cdot \mathsf{poly}(\lambda)$.

Remark 4 ($(1+\eta)$ blowup in round complexity)**.** *The round complexity blowup in the above compiler is constant, namely, an input protocol that consists of R rounds will result with a maliciously secure protocol with $O(R)$ rounds, where the constant underlying the O depends on the ratio between several parameters of the system, which could be large in concrete instantiations. Here, we propose an optimization to reduce the blow up to $(1+\eta)$ for any fixed $\eta > 0$.*

In the current protocol, after every round we perform a "certification phase" which guarantees that the state of all machines is consistent and all parties acted honestly given the previous state. This phase costs some constant number of additional rounds, denoted c. Instead, we could do this phase only once every c/η rounds, which we refer to as an epoch. Within an epoch, each party would prove honest behaviour only to the machines it communicates with using zkSNARKs (i.e., each message will be accompanied with a SNARK attesting to the correct computation of the current message given the previous state and the incoming messages). In the proof, extraction will be possible in polynomial time within an epoch since it consists of only a constant number of rounds. Overall, the round complexity of the compiled protocol will be $(1+\eta)R$.

8.4 Security of the Compiler

In this section, we show that for any P2P semi-malicious secure protocol Π, the compiled protocol Π satisfies the definition of malicious security given in Definition 14. We do this by exhibiting a nonuniform polynomial-time simulator $\mathcal{S}$ for any nonuniform polynomial-time adversary $\mathcal{A}$ for which the experiments $\mathsf{real}^{\Pi}_{\mathcal{A}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ and $\mathsf{ideal}^{\mathcal{F}f}_{\mathcal{S}}(1^\lambda, 1^M, \{x_i\}_{i\in[M]})$ are computationally indistinguishable.

Let $\mathcal{A}$ be an arbitrary nonuniform polynomial-time adversary. We define the simulator with respect to $\mathcal{A}$ as follows.

Simulator 1 Malicious Protocol Simulator

Input : The simulator receives as input the security parameter 1^λ.

Simulated Setup: The simulator generates the setup for the corrupted parties as follows.

1. The simulator initializes $\mathcal{A}$ with 1^λ and queries $\mathcal{A}$ for the input size N.
2. The simulator generates two threshold signature scheme setups $(\mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]}) \leftarrow \mathsf{Sig.Setup}(1^\lambda, 1^M)$ using the honest aggregated signature setup algorithm.
3. The simulator chooses a hash function $h \leftarrow \mathsf{H.Setup}(1^\lambda, 1^s)$.
4. The simulator generates the simulated idse-zkSNARK crs $(\widetilde{crs}, \mathsf{td}) \leftarrow \Pi.\mathsf{Sim}_1(1^\lambda)$ using the simulated SNARK setup algorithm, then for each $i \in [M] \setminus \mathcal{C}$ generates the id-specific trapdoor $\mathsf{td}_i \leftarrow \Pi.\mathsf{Sim}_2(\widetilde{crs}, \mathsf{td}, (i, 0))$. It saves $\{\mathsf{td}_i\}_{i\in[M]\setminus\mathcal{C}}$.
5. The simulator generates the simulated public key for the underlying P2P semi-malicious protocol: $(\mathsf{smpk}, \mathsf{st}_{sim}) \leftarrow \mathsf{Sim.Setup}_1(1^\lambda, 1^M)$

6. The simulator sends $(\mathsf{vk}, \mathsf{vk}_{rcv}, h, \widetilde{crs}, \mathsf{smpk})$ to the adversary.

7. Upon receiving the set $\mathcal{C}$ of corrupted parties from the adversary, the simulator computes $(\{\mathsf{smsk}_i\}_{i\in\mathcal{C}}, \mathsf{st}'_{sim}) \leftarrow \mathsf{Sim.Setup}_2(\mathsf{st}_{sim}, \mathcal{C})$ and sets $\mathsf{st}_{sim} \leftarrow \mathsf{st}'_{sim}$. The simulator also receives x_i

8. The simulator responds to $\mathcal{A}$ with $\{(\mathsf{ssk}_i, \mathsf{ssk}_{rcv,i}, \mathsf{smsk}_i)\}_{i\in\mathcal{C}}$.

Commitment Phase Extraction: The simulator does the following to extract the corrupted parties' inputs during the commitment phase.

1. On behalf of each honest party P_i, compute two commitments $c_{k_i} \leftarrow \mathsf{C.Commit}(0; \alpha_{k_i})$ and $c_{\mathsf{st}_{i,0}} \leftarrow \mathsf{C.Commit}((0,0); \alpha_{\mathsf{st}_{i,0}})$, where both α_{k_i} and $\alpha_{\mathsf{st}_{i,0}}$ are uniform random.

2. Run the subprotocol $\mathsf{CalcMerkleTree}(h, \{c_{\mathsf{st}_{i,0}}||c_{k_i}\}_{i\in[M]})$ on behalf of all parties, interacting with $\mathcal{A}$ to obtain the messages of the corrupted parties, to obtain τ_0, the Merkle root of the transcript, along with $\theta_{\mathsf{st}_{i,0}}$, an opening to $c_{\mathsf{st}_{i,0}}||c_{k_i}$ with respect to τ_0, for each party P_i, $i \in [M] \setminus \mathcal{C}$. If for some i the opening is not valid, then force the next step to abort by refusing to respond on behalf of P_i.

3. Run the subprotocol $\mathsf{Agree}((0,\tau_0), \mathsf{vk}, \{\mathsf{ssk}_i\}_{i\in[M]})$ on behalf of the parties, using $\mathcal{A}$ to obtain the corrupted parties' messages, and abort if all honest parties do not obtain a consistent τ_0. If the subprotocol causes any honest party to abort then force the protocol to abort by refusing to respond on behalf of that party.

4. On behalf of each honest party P_i, calculate a simulated SNARK $\pi_{i,0} \leftarrow \Pi.\mathsf{Sim}_3(\widetilde{crs}, (i,0), \mathsf{td}_i, \Phi(i,0,0,\perp,\tau_0); \alpha_{snark,i,0})$.

5. The simulator runs $\mathsf{RecCompAndVerify}(\widetilde{crs}, \perp, \tau_0, \{\pi_{i,0}\}_{i\in[M]})$ on behalf of all parties, interacting with $\mathcal{A}$ to obtain the corrupted parties' messages, so that either every honest party obtains π_0, or at least one honest party aborts. If the subprotocol causes any honest party to abort then force the protocol to abort by refusing to respond on behalf of that party.

6. Let $\mathcal{E}_0$ be the extractor guaranteed by Lemma 8.2 with respect to the machine $\mathcal{A}_{ext,0}$ (defined below). Run $\mathcal{E}_0(0, \alpha_{snark}, \alpha_{sig}, \alpha_{\mathcal{A}_{ext,0}}, \{\alpha_{snark,i,0}\}_{i\in[M]\setminus\mathcal{C}})$, where $\alpha_{\mathcal{A}_{ext,0}}$ is the randomness for $\mathcal{A}_{ext,0}$ and is chosen such that $\mathcal{A}_{ext,0}$ reproduces the same behavior as that of the simulator and the adversary up to this point. $\mathcal{E}_0$ should output $\{(c_{\mathsf{st}_{i,0}}, \theta_{i,0}, \mathsf{st}_{i,0} = (x_i, r_i), k_i, \alpha_{k_i})\}_{i\in\mathcal{C}}$. If $\mathcal{E}_0$ fails, the simulator aborts.

7. Compute a Merkle root τ'_0 using the values $(x_i, r_i, k_i, \alpha_{k_i})$ extracted in the previous step along with the values $(0,0,0,\alpha_{k_i})$ committed to on behalf of the honest parties. If $\tau_0 \neq \tau'_0$ then the simulator aborts.

Evaluation Phase: The simulator simulates the evaluation phase as follows. For each of the R steps corresponding to rounds of the underlying protocol, the simulator proceeds as follows:

- **During round r:** The simulator starts the state st_{sim} for the underlying protocol simulator, along with a state $(\mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1}, \mathsf{msg}^{\mathsf{out}}_{i,r-1})$ for each corrupted P_i, a Merkle root τ_{r-1} for the previous round's global state, and an opening $\theta_{i,r-1}$ for $c_{\mathsf{st}_{i,r-1}}||\mathsf{msg}^{\mathsf{in}}_{i,r-1}||\mathsf{msg}^{\mathsf{out}}_{i,r-1}||c_{k_i}$ for each corrupted P_i with respect to τ_{r-1}. Here $c_{\mathsf{st}_{i,r-1}} = \mathsf{C.Commit}(\mathsf{st}_{i,r-1}; \mathsf{PRF}_{k_i})$, a commitment to P_i's private state. Finally, the simulator knows $(x_i, r_i, k_i, \alpha_{k_i})$ for each $i \in \mathcal{C}$ and c_{k_i}, α_{k_i} for $i \in [M] \setminus \mathcal{C}$ from the commitment phase extraction. The simulator does the following:

1. Compute the messages and state of the simulator for the underlying protocol:
   - If $r = 1$, compute $(\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i \in [M] \setminus \mathcal{C}}, \mathsf{st}'_{sim}) \leftarrow \mathsf{Sim.Round}_1(\mathsf{st}_{sim})$.
   - If $r > 1$, compute $(\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i \in [M] \setminus \mathcal{C}}, \mathsf{st}'_{sim}) \leftarrow \mathsf{Sim.Round}_r(\{(\mathsf{msg}^{\mathsf{out}}_{i,r-1}, x_i, r_i)\}_{i \in \mathcal{C}}, \mathsf{st}_{sim})$. If $\mathsf{Sim.Round}_r$ sends a query, forward the query to the ideal functionality, and forward the response back to $\mathsf{Sim.Round}_r$.
2. Set $\mathsf{st}_{sim} \leftarrow \mathsf{st}'_{sim}$.
3. For each $i \in [M] \setminus \mathcal{C}$, choose $\alpha_{\mathsf{st}_{i,r}}$ uniformly at random, and set $c_{\mathsf{st}_{i,r}} \leftarrow \mathsf{C.Commit}(0; \alpha_{\mathsf{st}_{i,r}})$.
4. Send all honest parties' simulated messages to $\mathcal{A}$. For all $i \in [M] \setminus \mathcal{C}$:
   - For each $(j, s_j, e_j) \in \mathsf{OutgoingMessageLocs}(i, r)$, if $j \in \mathcal{C}$ then send $(j, \mathsf{msg}^{\mathsf{out}}_{i,r}[s_j : e_j])$ to $\mathcal{A}$.
5. Initialize an empty string $\mathsf{msg}^{\mathsf{in}}_{i,r}$ to the appropriate size for each honest party P_i.
6. Receive messages from corrupted parties to honest parties for round r from $\mathcal{A}$, and simulate messages between honest parties. For each $i \in [M] \setminus \mathcal{C}$:
   - For each message m received from $\mathcal{A}$ on behalf of P_j in round r, write m to $\mathsf{st}'_{i,r}$ at location $\mathsf{IncomingMessageLoc}(j, i, r)$.
   - For each $(j, s_j, e_j) \in \mathsf{OutgoingMessageLocs}(i, r)$, if $j \in [M] \setminus \mathcal{C}$ then copy $\mathsf{msg}^{\mathsf{out}}_{i,r}[s_j : e_j]$ to $\mathsf{msg}^{\mathsf{in}}_{j,r}$ at position (s, e), where $(s, e) \leftarrow \mathsf{IncomingMessageLoc}(i, j, r)$.
7. Run $\mathsf{CalcMerkleTree}(h, \{c_{\mathsf{st}_{i,r}} || \mathsf{msg}^{\mathsf{in}}_{i,r} || \mathsf{msg}^{\mathsf{out}}_{i,r} || c_{k_i}\}_{i \in [M]})$ on behalf of all parties, interacting with $\mathcal{A}$ to obtain the messages of the corrupted parties, to obtain τ_r, the Merkle root of the transcript, along with $\theta_{i,r}$, an opening to $c_{\mathsf{st}_{i,r}} || \mathsf{msg}^{\mathsf{in}}_{i,r} || \mathsf{msg}^{\mathsf{out}}_{i,r} || c_{k_i}$ with respect to τ_r, for each party P_i, $i \in [M] \setminus \mathcal{C}$. If for some i the opening is not valid, then force the next step to abort by refusing to respond on behalf of P_i.
8. Run $\mathsf{Agree}((r, \tau_r), \mathsf{vk}, \{\mathsf{ssk}_i\}_{i \in [M]})$ on behalf of all parties, interacting with $\mathcal{A}$ to obtain the messages of the corrupted parties. Abort if all honest parties do not obtain a consistent τ_0. If the subprotocol causes any honest party to abort then force the protocol to abort by refusing to respond on behalf of that party.
9. For each $i \in [M] \setminus \mathcal{C}$:
   (a) For each corrupted party P_j which sent a message to P_i in round r, send $\theta_{m_{i,r}}$, an opening to position $\mathsf{IncomingMessageGlobalLoc}(j, i, r)$ in τ_r, to $\mathcal{A}$.
   (b) Receive openings sent to honest parties from $\mathcal{A}$ on behalf of corrupted parties. If $\mathcal{A}$ sends an opening to P_i which is invalid, or if it shows an honest party's message wasn't copied to a corrupted party's state, force the next phase to abort by refusing to respond on behalf of P_i.
   (c) Calculate a simulated SNARK $\pi_{r,i} \leftarrow \Pi.\mathsf{Sim}_3(\widetilde{crs}, (i, 0), \mathsf{td}_i, \Phi(i, r, 0, \tau_{r-1}, \tau_r))$.
10. The simulator runs $\mathsf{RecCompAndVerify}(\widetilde{crs}, \tau_{r-1}, \tau_r, \{\pi_{i,r}\}_{i \in [M]})$ on behalf of all parties, interacting with $\mathcal{A}$ to obtain the corrupted parties' messages, so that either every honest party obtains a valid proof π_0, or at least one honest party aborts. If the subprotocol causes any honest party to abort then force the protocol to abort by refusing to respond on behalf of that party.

11. Compute what the corrupted parties' messages and committed states should be according an honest protocol execution with respect to the extracted inputs and randomness (x_i, r_i), the PRF keys k_i used as randomness for the commitments, and the honest parties' messages up to this point. For each $i \in \mathcal{C}$:
    (a) Compute $(\hat{\mathsf{st}}'_{i,r}, \hat{\mathsf{msg}}^{\mathsf{out}}{}_{i,r}) \leftarrow \mathsf{NextSt}_{i,r}(\mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$.
    (b) Initialize $\hat{\mathsf{msg}}^{\mathsf{in}}{}_{i,r}$ as an empty string of the appropriate size.
    (c) For each $(j, s_j, e_j) \in \mathsf{OutgoingMessageLocs}(i, r)$, if $j \in \mathcal{C}$, let $m = \hat{\mathsf{msg}}^{\mathsf{out}}{}_{i,r}[s_j : e_j]$ and write m to $\hat{\mathsf{msg}}^{\mathsf{in}}{}_{j,r}$ at location $\mathsf{IncomingMessageLoc}(i, j, r)$. If $j \in [M] \setminus \mathcal{C}$, let $m = \hat{\mathsf{msg}}^{\mathsf{out}}{}_{i,r}[s_j : e_j]$ and write the honest-behavior message m to $\mathsf{msg}^{\mathsf{in}}_{j,r}$ at location $\mathsf{IncomingMessageLoc}(i, j, r)$, overwriting the adversary's message.
    (d) For each $i \in [M] \setminus \mathcal{C}$:
        – For each $(j, s_j, e_j) \in \mathsf{OutgoingMessageLocs}(i, r)$, if $j \in \mathcal{C}$ then set $m = (j, \mathsf{msg}^{\mathsf{out}}_{i,r}[s_j : e_j])$, and write m to $\hat{\mathsf{msg}}^{\mathsf{in}}{}_{j,r}$ at location $\mathsf{IncomingMessageLoc}(i, j, r)$.
    (e) Compute $\hat{c}_{\mathsf{st}_{i,r}} \leftarrow \mathsf{C.Commit}(\mathsf{st}_{i,r}; \mathsf{PRF}_{k_i}(r))$.
12. Compute a Merkle root $\hat{\tau}_0$ using the values $(\hat{c}_{\mathsf{st}_{i,r}}, \hat{\mathsf{msg}}^{\mathsf{in}}{}_{i,r}, \hat{\mathsf{msg}}^{\mathsf{out}}{}_{i,r}, c_{k_i})$ on behalf of the corrupted parties along with the simulated states and commitments to k_i $(c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{in}}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r}, c_{k_i})$ on behalf of the honest parties. If $\tau_0 \neq \hat{\tau}_0$ then the simulator aborts.

Output Delivery: If the last round finishes, the simulator delivers the output to all honest parties which did not abort.

$\mathcal{A}_{ext,0}(1^\lambda, 1^M)$:

Input: Security parameter 1^λ, number of machines 1^M.

Hardcoded: adversary $\mathcal{A}$.

Behavior: $\mathcal{A}_{ext,r}$ enacts the simulation strategy defined above with $\mathcal{A}$ up to the end of the $\mathsf{RecCompAndVerify}$ subprotocol in the commitment phase, except for the following differences. During the generation of the setup parameters:

1. During the first part of the setup generation, instead of running the idse-zkSNARK setup directly $\mathcal{A}'$ receives $\widetilde{crs}$ from the challenger and uses it when forwarding the public setup parameters to the adversary during the setup phase. It generates the rest of the setup parameters as normal.
2. Once $\mathcal{A}'$ receives the corrupted party set $\mathcal{C}$ from $\mathcal{A}$, it forwards this to the challenger.

During the commitment phase:

1. After the Agree subprotocol, when all parties have agreed on τ_0, $\mathcal{A}'$ sends τ_0 to the challenger.
2. During the $\mathsf{RecCompAndVerify}$ subprotocol, $\mathcal{A}'$ obtains the honest parties' messages from the challenger and forwards them to $\mathcal{A}$, and receives the corrupted parties' messages from $\mathcal{A}$ and forwards them to the challenger.

$\mathcal{A}_{ext,0}$ halts after the end of the $\mathsf{RecCompAndVerify}$ subprotocol.

8.4.1 Proof of Indistinguishability

We prove that no adversary $\mathcal{A}$ can distinguish between a real-world execution with honest parties and an ideal-world execution with the simulator defined in Section 8.4. We do this by exhibiting a sequence of hybrids, each describing an interaction between $\mathcal{A}$ and a simulator, as follows:

Hyb_0: In this hybrid, the simulator enacts a real-world execution with $\mathcal{A}$, taking the part of the honest parties.

Hyb_1: In this hybrid, the simulator behaves identically to Hyb_0 except regarding the idse-zkSNARK. During setup, it uses the $\Pi.\mathsf{Sim}_1$ and $\Pi.\mathsf{Sim}_2$ algorithms to generate the setup and trapdoors, and uses $\Pi.\mathsf{Sim}_3$ to simulate every idse-zkSNARK generated by the honest parties.

Hyb_2: In this hybrid, the simulator behaves identically to Hyb_1 except that during the commitment phase, on behalf of each honest party P_i, it computes the commitment c_{k_i} as a commitment to 0 instead of a commitment to an honest PRF key k_i.

Hyb_3: In this hybrid, the simulator behaves identically to Hyb_2 except that when computing the commitment $c_{\mathsf{st}_{i,r}}$ on behalf of each honest party P_i during the commitment phase and during each round r, it uses a uniform random string as the source of randomness instead of randomness chosen by PRF_{k_i}.

Hyb_4: In this hybrid, the simulator behaves identically to Hyb_3 except that during the commitment phase and during each round r, on behalf of each honest party P_i, it computes the commitment $c_{\mathsf{st}_{i,r}}$ as a commitment to 0 instead of a commitment to the honest state $\mathsf{st}_{i,r}$ (or instead of to (x_i, r_i) in the commitment phase).

Hyb_5: In this hybrid, the simulator behaves identically to Hyb_4 except that after running the Agree subprotocol during the commitment phase and during each round r, it aborts if every honest party does not output a consistent Merkle root τ_r.

Hyb_6: In this hybrid, the simulator behaves identically to Hyb_5 except that after the $\mathsf{RecCompAndVerify}$ subprotocol in the commitment phase, the simulator runs the extractor $\mathcal{E}_0$ with respect to the machine $\mathcal{A}_{ext,0}$, and aborts if the extractor fails (see step 6 in the simulation strategy above). It then uses the extracted values to recompute the Merkle root for the commitment phase, and aborts if this Merkle root is not equal to the Merkle root computed by the parties (see step 7 in the simulation strategy above).

Hyb_7: In this hybrid, the simulator behaves identically to Hyb_6 except that during each round r after the $\mathsf{RecCompAndVerify}$ subprotocol, the simulator runs steps 11 and 12 in the simulation strategy above. That is, during each round, it uses the values extracted by $\mathcal{E}_0$ in the commitment phase to maintain states $\{\hat{\mathsf{st}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}\}_{i\in\mathcal{C}}$ for all corrupted parties according an honest execution of the protocol. Then, for each round the simulator uses $\{\hat{\mathsf{st}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}||c_{k_i}, k_i\}_{i\in\mathcal{C}}$ along with the simulated honest parties' $\{c_{\mathsf{st}_{i,r}}||\mathsf{msg}^{\mathsf{in}}_{i,r}||\mathsf{msg}^{\mathsf{out}}_{i,r}||c_{k_i}\}_{i\in[M]\setminus\mathcal{C}}$ to recompute the Merle root for this round, and aborts if this Merkle root is not equal to the one computed by the parties.

Hyb_8: In this hybrid, the simulator behaves identically to Hyb_7 except that instead of computing the honest parties' behavior for each P2P semi-malicious round according to the honest protocol specification, it uses the P2P semi-malicous simulator to determine their behavior. The behavior of the simulator in this hybrid is now identical to that of the ideal world.

Claim 1. *Assuming the zero-knowledge property of the idse-zkSNARK holds, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_0 *and* Hyb_1*.*

Proof. The only difference between Hyb_0 and Hyb_1 is that during the setup the simulator generates a simulated SNARK setup, and during the commitment phase and during the phase for each round r, the simulator computes simulated idse-zkSNARKs instead of honest ones. Assume that $\mathcal{A}$ distinguishes between Hyb_0 and Hyb_1 with non-negligible probability. We build a reduction $\mathcal{A}'$ to the multi-theorem zero-knowledge property of the idse-zkSNARK scheme.

$\mathcal{A}'$ is an oracle machine which has $\mathcal{A}$ hardcoded and takes in a value crs. $\mathcal{A}'$ runs the Hyb_0 experiment with $\mathcal{A}$ except for the following differences.

- During the setup phase, instead of generating the idse-zkSNARK crs itself, $\mathcal{A}'$ uses the value crs which was given as input to $\mathcal{A}'$.
- During the commitment phase and during the phase for each round r, instead of computing each honest party P_i's proof $\pi_{i,r}$ itself, $\mathcal{A}'$ calls the oracle with values $(\Phi(i, r, 0, \tau_{r-1}, \tau_r), w)$, where w is the valid witness for the statement $\Phi(i, r, 0, \tau_{r-1}, \tau_r)$. $\mathcal{A}'$ then uses the proof $\pi_{i,r}$ it receives as P_i's proof during the $\mathsf{RecCompAndVerify}$ protocol for that round.

If we are in the real world of the idse-zkSNARK zero-knowledge game, then $\mathcal{A}'$ enacts exactly Hyb_0. On the other hand, if we are in the ideal world, then $\mathcal{A}'$ enacts Hyb_1. Thus, since $\mathcal{A}$ distinguishes between these hybrids, $\mathcal{A}'$ successfully beats the zero-knowledge game of the idse-zkSNARK scheme with non-negligible probability. □

Claim 2. *Assuming the hiding property of the noninteractive commitment scheme holds, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_1 *and* Hyb_2*.*

Proof. The only difference between Hyb_1 and Hyb_2 is that in Hyb_2, during the commitment phase, the simulator computes a commitment $c_{k_i} \leftarrow \mathsf{C.Commit}(0)$ instead of $c_{k_i} \leftarrow \mathsf{C.Commit}(k_i)$. We prove indistinguishability of these two hybrids via a sequence of subhybrids $\mathsf{Hyb}_{1,0}, \ldots, \mathsf{Hyb}_{1,M}$, where $\mathsf{Hyb}_{1,0} = \mathsf{Hyb}_1$ and $\mathsf{Hyb}_{1,M} = \mathsf{Hyb}_2$. We define the hybrids as follows:

> $\mathsf{Hyb}_{1,i}$: In this hybrid, the simulator behaves identically to Hyb_1 except that whenever $i' \leq i$, the simulator computes $c_{k_{i'}} \leftarrow \mathsf{C.Commit}(0)$ on behalf of $P_{i'}$ instead of $c_{k_{i'}} \leftarrow \mathsf{C.Commit}(k_{i'})$.

We prove indistinguishability between each successive pair of hybrids $\mathsf{Hyb}_{1,i-1}$ and $\mathsf{Hyb}_{1,i}$. Assume $\mathcal{A}$ distinguishes between some pair of hybrids with non-negligible probability. Fix the simulator's choice of k_i which maximizes the probability that $\mathcal{A}$ succeeds in distinguishing. We construct a reduction $\mathcal{A}'$ to the hiding property of the commitment scheme.

We define $\mathcal{A}'$ as follows. $\mathcal{A}'$ is machine which has $\mathcal{A}$ hardcoded and interacts with a challenger for the hiding game of the commitment scheme. $\mathcal{A}'$ runs the first hybrid experiment with $\mathcal{A}$ except for the following difference. During the commitment phase, $\mathcal{A}'$ receives a commitment c from the challenger, which either commits to k_i or 0. $\mathcal{A}'$ then uses this commitment as c_{k_i}.

If the challenger responds with a commitment to c_{k_i} then $\mathcal{A}'$'s behavior is exactly that of the first hybrid. On the other hand, if the challenger responds a commitment to 0, then $\mathcal{A}'$'s behavior is that of the second hybrid. Thus, since $\mathcal{A}$ distinguishes between these hybrids, $\mathcal{A}'$ successfully beats the commitment scheme hiding game with non-negligible probability. □

Claim 3. *Assuming the pseudorandomness property of the PRF holds, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_2 *and* Hyb_3*.*

Proof. The only difference between Hyb_2 and Hyb_3 is that in Hyb_3 the simulator generates commitments on behalf of the honest parties using true uniform randomness instead of the pseudorandom values used in Hyb_2. We prove indistinguishability of these two hybrids via a sequence of subhybrids $\mathsf{Hyb}_{2,0,0}, \ldots, \mathsf{Hyb}_{2,0,M}, \mathsf{Hyb}_{2,1,1}, \ldots, \mathsf{Hyb}_{2,R,M}$, where $\mathsf{Hyb}_{2,0,0} = \mathsf{Hyb}_2$ and $\mathsf{Hyb}_{2,R,M} = \mathsf{Hyb}_3$. We define the hybrids as follows:

> $\mathsf{Hyb}_{2,r,i}$: In this hybrid, the simulator behaves identically to Hyb_2 except that whenever $r' \leq r$ and $i' \leq i$, when computing the commitment $c_{\mathsf{st}_{i',r'}}$ on behalf of each honest party $P_{i'}$ during the commitment phase and during each round r', it uses a uniform random string as the source of randomness instead of randomness chosen by PRF_{k_i}.

We prove indistinguishability between each successive pair of hybrids $\mathsf{Hyb}_{2,r,i-1}$ (or $\mathsf{Hyb}_{2,r-1,M}$) and $\mathsf{Hyb}_{2,r,i}$. Assume $\mathcal{A}$ distinguishes between some pair of hybrids with non-negligible probability. We construct a reduction $\mathcal{A}'$ to the PRF game.

We define $\mathcal{A}'$ as follows. $\mathcal{A}'$ is machine which has $\mathcal{A}$ hardcoded and interacts with a challenger for the PRF game. $\mathcal{A}'$ runs the first hybrid experiment with $\mathcal{A}$ except for the following difference. During round r when computing the commitment $\mathsf{C.Commit}(\mathsf{st}_{i,r}; \alpha)$ on behalf of party P_i, it queries the challenger for a string and uses this string as the randomness α for the $\mathsf{C.Commit}$ algorithm.

If the challenger responds with a PRF evaluation then $\mathcal{A}'$'s behavior is exactly that of the first hybrid. On the other hand, if the challenger responds with a truly random string, then $\mathcal{A}'$'s behavior is that of the second hybrid. Thus, since $\mathcal{A}$ distinguishes between these hybrids, $\mathcal{A}'$ successfully beats the PRF indistinguishability game with non-negligible probability. □

Claim 4. *Assuming the hiding property of the noninteractive commitment scheme holds, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_3 *and* Hyb_4.

Proof. The only difference between Hyb_3 and Hyb_4 is that in Hyb_4, for each $i \in [M] \setminus \mathcal{C}$ and $r \in 0, \ldots, R$, the simulator computes $c_{\mathsf{st}_{i,r}}$ as a commitment to 0 instead of a commitment to the honest state $\mathsf{st}_{i,r}$ (or instead of to (x_i, r_i) in the commitment phase). We prove indistinguishability of these two hybrids via a sequence of subhybrids $\mathsf{Hyb}_{3,0,0}, \ldots, \mathsf{Hyb}_{3,0,M}, \mathsf{Hyb}_{3,1,1}, \ldots, \mathsf{Hyb}_{3,R,M}$, where $\mathsf{Hyb}_{3,0,0} = \mathsf{Hyb}_3$ and $\mathsf{Hyb}_{3,R,M} = \mathsf{Hyb}_4$. We define the hybrids as follows:

> $\mathsf{Hyb}_{3,r,i}$: In this hybrid, the simulator behaves identically to Hyb_1 except that whenever $r' \leq r$ and $i' \leq i$, the simulator computes $c_{\mathsf{st}_{i,r}}$ as a commitment to 0 instead of a commitment to the honest state $\mathsf{st}_{i,r}$ (or instead of to (x_i, r_i) in the commitment phase).

We prove indistinguishability between each successive pair of hybrids $\mathsf{Hyb}_{3,r,i-1}$ (or $\mathsf{Hyb}_{3,r-1,M}$) and $\mathsf{Hyb}_{3,r,i}$. Assume $\mathcal{A}$ distinguishes between some pair of hybrids with non-negligible probability. Fix the randomness used by $\mathcal{A}$ as well as the randomness used by the simulator except for the randomness used in committing to $\mathsf{st}_{i,r}$ which maximizes the probability of $\mathcal{A}$ winning the distinguishing game. This fixes the state $\mathsf{st}_{i,r}$ of honest party P_i during round r. We construct a reduction $\mathcal{A}'$ to the hiding property of the commitment scheme.

We define $\mathcal{A}'$ as follows. $\mathcal{A}'$ is machine which has $\mathcal{A}$ hardcoded and interacts with a challenger for the hiding game of the commitment scheme. $\mathcal{A}'$ runs the first hybrid experiment with $\mathcal{A}$ except for the following difference. During round r, $\mathcal{A}'$ receives a commitment c from the challenger, which either commits to $\mathsf{st}_{i,r}$ or 0. $\mathcal{A}'$ then uses this commitment as $c_{\mathsf{st}_{i,r}}$.

If the challenger responds with a commitment to $\mathsf{st}_{i,r}$ then then $\mathcal{A}'$ behaves in the same way as the first hybrid. On the other hand, if the challenger responds with a commitment to 0, then $\mathcal{A}'$ has the behavior of the second hybrid. Thus, since $\mathcal{A}$ distinguishes between these hybrids with non-negligible probability, $\mathcal{A}'$ successfully beats the hiding game of the commitment scheme with non-negligible probability. □

Claim 5. *Assuming unforgeability of the threshold signature scheme, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_4 *and* Hyb_5.

Proof. We prove this claim via Lemma 8.1. Assume that there is an adversary $\mathcal{A}$ which can distinguish between the two hybrids Hyb_4 and Hyb_5 with non-negligible probability. We build a reduction $\mathcal{A}'$ against the game $\mathsf{AgreeSecurity}$ as follows.

$\mathcal{A}'$ has the original adversary $\mathcal{A}$ hardcoded and interacts with the $\mathsf{AgreeSecurity}$ challenger. It enacts Hyb_5 with $\mathcal{A}$ except for the following differences. During the generation of the setup parameters:

1. During the first part of the setup generation, $\mathcal{A}'$ receives the TSDR verification key vk from the challenger and uses it when forwarding the public setup parameters to the adversary during the Hyb_5 game. It generates the rest of the setup parameters as normal.

2. Once $\mathcal{A}'$ receives the corrupted party set $\mathcal{C}$ from $\mathcal{A}$, it forwards this to the challenger to receive the TSDR secret keys $\{\mathsf{ssk}\}_{i\in\mathcal{C}}$, and forwards these secret keys to the adversary as part of the private setup parameters.

During the commitment phase and during every round r:

1. After the $\mathsf{CalcMerkleTree}$ subprotocol, for each honest party P_i, $\mathcal{A}'$ sends $x_i = \tau_{r,i}$ to the challenger. Here τ_{r_i} is the output of $\mathsf{CalcMerkleTree}$ according to P_i.

2. During the Agree subprotocol, $\mathcal{A}'$ obtains the honest parties' messages from the challenger and forwards them to $\mathcal{A}$, and receives the corrupted parties' messages from $\mathcal{A}$ and forwards them to the challenger.

The only difference between Hyb_4 and Hyb_5 is that in Hyb_5, during each round r after the Agree subprotocol, the simulator aborts if all honest parties fail to agree on (r, τ_r). Thus, since $\mathcal{A}$ successfully distinguishes between the two hybrids with non-negligible probability, this means that $\mathcal{A}$ is able to induce such an abort during Hyb_5. Because $\mathcal{A}'$ behaves identically to Hyb_5 when we run $\mathsf{AgreeSecurity}_{\mathcal{A}'}(1^\lambda, R)$, we have that $\Pr\left[\mathsf{AgreeSecurity}_{\mathcal{A}'}(1^\lambda, R)\right]$ is non-negligible, contradicting Lemma 8.1. □

Claim 6. *Assuming unforgeability of the threshold signature scheme and id-based simulation-extractability of the idse-zkSNARK, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_5 *and* Hyb_6.

Proof. We prove this claim via Lemma 8.2. Assume that there is an adversary $\mathcal{A}$ which can distinguish between the two hybrids Hyb_5 and Hyb_6 with non-negligible probability. We build a reduction $\mathcal{A}'$ against the game $\mathsf{RCVSecurity}$ as follows.

$\mathcal{A}'$ has the original adversary $\mathcal{A}$ hardcoded and interacts with the $\mathsf{RCVSecurity}$ challenger. It enacts Hyb_6 with $\mathcal{A}$ except for the following differences. During the generation of the setup parameters:

1. During the first part of the setup generation, $\mathcal{A}'$ receives the idse-zkSNARK setup $\widetilde{crs}$ from the challenger and uses it as the CRS in the Hyb_6 game. It generates the rest of the setup parameters as normal.

2. Once $\mathcal{A}'$ receives the corrupted party set $\mathcal{C}$ from $\mathcal{A}$, it forwards this to the challenger.

During the commitment phase and during every round r:

1. After the Agree subprotocol, when all parties have agreed on τ_r, $\mathcal{A}'$ sends τ_r along with τ_{r-1} (or just τ_0 if it is the commitment phase) to the challenger.

2. During the $\mathsf{RecCompAndVerify}$ subprotocol, $\mathcal{A}'$ obtains the honest parties' messages from the challenger and forwards them to $\mathcal{A}$, and receives the corrupted parties' messages from $\mathcal{A}$ and forwards them to the challenger.

The only difference between Hyb_5 and Hyb_6 is that in Hyb_6, after the $\mathsf{RecCompAndVerify}$ subprotocol, the simulator runs the extractor $\mathcal{E}_0$ guaranteed by Lemma 8.2 and aborts if the extractor fails or if the extracted values fail to reconstruct τ_0. Thus, since $\mathcal{A}$ successfully distinguishes between the two hybrids with non-negligible probability, this means that $\mathcal{A}$ is able to induce such an abort during Hyb_6. Because $\mathcal{A}'$ behaves identically to Hyb_6 when we run $\mathsf{RCVSecurity}_{\mathcal{A}',\mathcal{E}_0}(1^\lambda, R, 0, \bot)$, we have that $\Pr\left[\mathsf{RCVSecurity}_{\mathcal{A}',\mathcal{E}_0}(1^\lambda, R, 0, \bot)\right]$ is non-negligible, contradicting Lemma 8.2. □

Claim 7. *Assuming simulation-extractability of the idse-zkSNARK, collision-resistance of the hash function and binding of the noninteractive commitment scheme, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_6 *and* Hyb_7.

Proof. We prove the claim via a sequence of subhybrids $\mathsf{Hyb}_{6,0}, \ldots, \mathsf{Hyb}_{6,R}$, where $\mathsf{Hyb}_{6,0}$ is identical to Hyb_6 and $\mathsf{Hyb}_{6,R}$ is identical to Hyb_7.

> $\mathsf{Hyb}_{6,r}$: Behave identically to Hyb_7 until (including) round r, and behave identically to Hyb_6 afterwards

Assume $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability. The only difference between the two hybrids is that in $\mathsf{Hyb}_{6,r}$ the simulator aborts if the new Merkle root computed by the $\mathsf{CalcMerkleTree}$ is different than the Merkle root of an honest execution of the corrupted parties based on the values extracted in the commitment phase. We finish the argument in two steps. First, we show that if it is possible to extract both the commitment-phase witnesses and the round-r witnesses used by the corrupted parties, then this yields an efficient algorithm which obtains two different openings for either a commitment or a Merkle root. Second, we show how to extract the witnesses.

Let $\mathsf{RCVSecurity}'$ be a variant of $\mathsf{RCVSecurity}$ which, when outputting 0, also outputs the witnesses for rounds r_1 and r_2. Assume there is an adversary $\mathcal{A}'$ which has $\mathcal{A}$ hardcoded, and where $\mathsf{RCVSecurity}_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(1^\lambda, R, r_1 = 0, r_2 = r)$ enacts $\mathsf{Hyb}_{6,r}$ and outputs 0 with overwhelming probability. This means $\mathsf{RCVSecurity}'_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(1^\lambda, R, r_1 = 0, r_2 = r)$ outputs a distribution of witnesses that corresponds exactly to the proofs of $\mathcal{A}$ in $\mathsf{Hyb}_{6,r}$. Thus, with non-negligible probability, $\mathsf{RCVSecurity}'_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(1^\lambda, R, r_1 = 0, r_2 = r)$ outputs two sets of valid witnesses $W_0 = \{(\hat{c}_{\mathsf{st}_{0,0}}, \hat{\theta}_{i,0}, \hat{k}_i, \hat{c}_{k_i}, \hat{\alpha}_{k_i}, \mathsf{st}_{i,0} = (x_i, r_i))\}_{i \in \mathcal{C}}$ and $W_r = \{(c_{\mathsf{st}_{i,r-1}}, \mathsf{msg}^{\mathsf{in}}_{i,r-1}, \mathsf{msg}^{\mathsf{out}}_{i,r-1}, \theta_{i,r-1}, c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{in}}_{i,r}, \mathsf{msg}^{\mathsf{out}}_{i,r}, \theta_{i,r}, k_i, c_{k_i}, \alpha_{k_i}, \mathsf{st}_{i,r-1}, \mathsf{st}_{i,r}, \{(m_{j,r}, \theta_{m_{j,r}})\}_j)\}_{i \in \mathcal{C}}$, one for the statement $\Phi(0, 0, t, \bot, \tau_0)$ for the commitment phase and one for the statement $\Phi(0, r, t, \tau_{r-1}, \tau_r)$ for round r, which correspond to an execution of $\mathsf{Hyb}_{6,r}$ where the simulator aborts during round r. Because of this, it holds that the Merkle root $\hat{\tau}_r$ computed by the simulator from the set of witnesses W_0 is not equal to the Merkle root τ_r computed by the parties during $\mathsf{CalcMerkleTree}$. Note however that $\hat{\tau}_{r-1} = \tau_{r-1}$, where $\hat{\tau}_{r-1}$ is the honest-behavior verification root calculated by the simulator during round $r - 1$ (since the simulator did not abort in round $r - 1$). Using the fact that $\hat{\tau}_r \neq \tau_r$ and $\hat{\tau}_{r-1} = \tau_{r-1}$, along with the valid openings for τ_r and τ_{r-1} in W_r, we show how to either obtain two openings for τ_{r-1} or two openings for one of the commitments $c_{\mathsf{st}_{i,r}}$.

Without loss of generality we can assume the difference in the strings committed to by the Merkle roots is at a position corresponding to some player P_i's state. In other words, letting $\hat{\mathsf{st}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}||\hat{c}_{k_i}$ be the substring committed to as part of $\hat{\tau}_r$ on behalf of P_i, we assume that

$$\hat{c}_{\mathsf{st}_{i,r}}||\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r}||\hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}||\hat{c}_{k_i} \neq c_{\mathsf{st}_{i,r}}||\mathsf{msg}^{\mathsf{in}}_{i,r}||\mathsf{msg}^{\mathsf{out}}_{i,r}||c_{k_i}.$$

There are several cases: either $i \in \mathcal{C}$ or $i \in [M] \setminus \mathcal{C}$, and either $(\hat{c}_{\mathsf{st}_{i,r}}, \hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}) \neq (c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{out}}_{i,r})$, $\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r} \neq \mathsf{msg}^{\mathsf{in}}_{i,r}$, or $\hat{c}_{k_i} \neq c_{k_i}$. At least one of these cases must occur with non-negligible probability. We handle each case as follows.

We start with the case where $i \in \mathcal{C}$. First, assume $\hat{c}_{k_i} \neq c_{k_i}$. This means that the simulator has an opening $\hat{\theta}_{i,r-1}$ to a value $\hat{c}_{k_i}$ in τ_{r-1} at the same location as the opening $\theta_{i,r-1}$ from W_r, which is to a different value c_{k_i} So if this case happens with non-negligible probability then we contradict the collision-resistance of the hash function.

Next, assume that $\hat{c}_{k_i} = c_{k_i}$ but $(\hat{c}_{\mathsf{st}_{i,r}}, \hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}) \neq (c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{out}}_{i,r})$. Letting $\hat{c}_{\mathsf{st}_{i,r-1}}$, $\hat{\mathsf{st}}_{i,r-1}$, $\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r-1}$, and $\hat{\mathsf{msg}^{\mathsf{out}}}_{i,r-1}$, and $\hat{k}_i$ be the values for P_i which the simulator computed during the honest behavior verification step in round $r-1$, this means that either $k_i \neq \hat{k}_i$ or $(\hat{\mathsf{st}}_{i,r-1}, \hat{\mathsf{msg}^{\mathsf{in}}}_{i,r-1}) \neq (\mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$. It is impossible for $k_i \neq \hat{k}_i$, because that would contradict perfect binding of the commitment scheme. But if $(\hat{\mathsf{st}}_{i,r-1}, \hat{\mathsf{msg}^{\mathsf{in}}}_{i,r-1}) \neq (\mathsf{st}_{i,r-1}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$, then we have that $(\hat{c}_{\mathsf{st}_{i,r-1}}, \hat{\mathsf{msg}^{\mathsf{in}}}_{i,r-1}) \neq (c_{\mathsf{st}_{i,r-1}}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$, also by perfect binding of the commitment scheme. The simulator has an opening $\hat{\theta}_{i,r-1}$ which opens $\tau_{i,r-1}$ to $(\hat{c}_{\mathsf{st}_{i,r-1}}, \hat{\mathsf{msg}^{\mathsf{in}}}_{i,r-1})$, whereas W_r contains $\theta_{i,r-1}$ which opens τ_{r-1} to $(c_{\mathsf{st}_{i,r-1}}, \mathsf{msg}^{\mathsf{in}}_{i,r-1})$ at the same location. So if this happens with non-negligible probability then we contradict the collision-resistance of the hash function.

For the final subcase when $i \in \mathcal{C}$, assume $(\hat{c}_{\mathsf{st}_{i,r}}, \hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}, \hat{c}_{k_i}) = (c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{out}}_{i,r}, c_{k_i})$ but $\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r} \neq \mathsf{msg}^{\mathsf{in}}_{i,r}$. Recall that $\mathsf{msg}^{\mathsf{in}}_{i,r}$ consists of messages $m_{j,i,r}$ sent from P_j to P_i during round r. Without loss of generality, assume $\hat{m}_{j,i,r} \neq m_{j,i,r}$, for a particular choice of j. Recall that the adversary must give as part of W_r an opening $\theta_{m_{j,i,r}}$ for $m_{j,i,r}$ in τ_r at the correct location inside $\mathsf{msg}^{\mathsf{out}}_{i,r}$. The simulator also has an opening for $\hat{m}_{j,i,r}$ at this same location in τ_r. Thus, if this case happens with non-negligible probability then we contradict the collision-resistance of the hash function.

Now we handle the case where $i \in [M] \setminus \mathcal{C}$. Without loss of generality, assume there is no $i \in \mathcal{C}$ where the substrings are different. We first observe that it cannot be the case that $i \in [M] \setminus \mathcal{C}$ and $(\hat{c}_{\mathsf{st}_{i,r}}, \hat{\mathsf{msg}^{\mathsf{out}}}_{i,r}) \neq (c_{\mathsf{st}_{i,r}}, \mathsf{msg}^{\mathsf{out}}_{i,r})$ or $\hat{c}_{k_i} \neq c_{k_i}$, since the simulator uses the same values at these positions for $\hat{\tau}_r$ and τ_r. So the only possibility for when $i \in [M] \setminus \mathcal{C}$ is if $\hat{\mathsf{msg}^{\mathsf{in}}}_{i,r} \neq \mathsf{msg}^{\mathsf{in}}_{i,r}$. If this is the case, then there must be some $j \in \mathcal{C}$ such that P_j sends a message to P_i during round r, and that the message $m_{j,i,r}$ sent by the adversary on behalf of P_j is not equal to the message $\hat{m}_{j,i,r}$ which the simulator determined should be sent by P_j during the honest behavior verification step of round r. This means that the adversary's message m is not equal to $\mathsf{msg}^{\mathsf{out}}_{j,r}[s_i : e_i]$, where $\mathsf{msg}^{\mathsf{out}}_{j,r}$ is from W_r and $(i, s_i, e_i) \in \mathsf{OutgoingMessageLocs}(j, r)$, since by assumption $\mathsf{msg}^{\mathsf{out}}_{j,r} = \hat{\mathsf{msg}^{\mathsf{out}}}_{j,r}$. Thus $\theta_{m_j,r}$ does not open τ_r at position $\mathsf{IncomingMessageGlobalLoc}(\lambda, j, i)$ to m. But the simulator has an opening $\theta_{i,r}$, which opens τ_r to m at position $\mathsf{IncomingMessageGlobalLoc}(\lambda, j, i)$, so if this case occurs with non-negligible probability then we have a contradiction of the collision-resistance of the hash function.

We finally show the existence of an $\mathcal{A}'$ satisfying the requirements above. Let $\mathcal{A}'$ be an adversary for the $\mathsf{RCVSecurity}$ game, defined as follows. $\mathcal{A}'$ has the original adversary $\mathcal{A}$ hardcoded and interacts with the $\mathsf{RCVSecurity}$ challenger. It enacts $\mathsf{Hyb}_{6,r}$ with $\mathcal{A}$ except for the following differences. During the generation of the setup parameters:

1. During the first part of the setup generation, $\mathcal{A}'$ receives the idse-zkSNARK setup $\widetilde{crs}$ from the challenger and uses it as the public setup parameters for the $\mathsf{Hyb}_{6,r}$ game. It generates the rest of the setup parameters as normal.
2. Once $\mathcal{A}'$ receives the corrupted party set $\mathcal{C}$ from $\mathcal{A}$, it forwards this to the challenger.

During the commitment phase and during every round r:

1. After the Agree subprotocol, when all parties have agreed on τ_r, $\mathcal{A}'$ sends τ_r along with τ_{r-1} (or just τ_0 if it is the commitment phase) to the challenger.
2. During the $\mathsf{RecCompAndVerify}$ subprotocol, $\mathcal{A}'$ obtains the honest parties' messages from the challenger and forwards them to $\mathcal{A}$, and receives the corrupted parties' messages from $\mathcal{A}$ and forwards them to the challenger.

By Lemma 8.2, $\mathsf{RCVSecurity}_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(1^\lambda, R, r_1 = 0, r_2 = r)$ outputs 0 with overwhelming probability, which means $\mathsf{RCVSecurity}_{\mathcal{A}',\mathcal{E}_{\mathcal{A}'}}(1^\lambda, R, r_1 = 0, r_2 = r)$ outputs witness sets W_0 and W_r which are valid for the corresponding execution of $\mathsf{Hyb}_{6,r}$ with overwhelming probability. We have now shown both steps of our argument, thus proving the claim. □

Claim 8. *Assuming P2P semi-malicious security of the underlying protocol, the view of the adversary and the outputs of the honest parties are indistinguishable in* Hyb_7 *and* Hyb_8.

Proof. The only difference between Hyb_7 and Hyb_8 is that in Hyb_7 the simulator computes the honest parties' messages using the P2P semi-malicious simulator, whereas in Hyb_8 it uses the honest protocol behavior. Assume $\mathcal{A}$ distinguishes between the two hybrids with non-negligible probability. We construct a reduction $\mathcal{A}'$ to the P2P semi-malicious security of the underlying MPC protocol.

We define $\mathcal{A}'$ as follows. $\mathcal{A}'$ has the adversary $\mathcal{A}$ hardcoded and interacts with the challenger for the P2P semi-malicious security game. Without loss of generality, assume that for round r the challenger sends the honest parties' messages in the form $\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}$, and then requires a response $\{(\mathsf{msg}^{\mathsf{out}}_{i,r}, x_i, r_i)\}_{i\in\mathcal{C}}$. $\mathcal{A}'$ enacts the Hyb_7 experiment with $\mathcal{A}$ except for the following differences. During generation of the setup parameters:

1. $\mathcal{A}'$ queries the challenger to receive smpk and forwards it to $\mathcal{A}$.
2. Once $\mathcal{A}$ replies with the corrupted set $\mathcal{C}$, $\mathcal{A}'$ forwards it to the challenger to receive back $\{\mathsf{smsk}_i\}_{i\in\mathcal{C}}$, which it forwards to $\mathcal{A}$.

During round r of the experiment:

1. Instead of computing the honest parties' messages directly, $\mathcal{A}'$ queries the challenger to obtain $\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}$, which it uses as the honest parties' outgoing messages.
2. After $\mathcal{A}'$ performs the honest-behavior verification step for round r, if Hyb_7 has not yet aborted, $\mathcal{A}'$ sends the set $\{(\mathsf{msg}^{\mathsf{out}}_{i,r}, x_i, r_i)\}_{i\in\mathcal{C}}$ to the challenger.

Note that $\mathcal{A}'$ is a valid P2P semi-malicious adversary, because the only way that Hyb_7 does not abort is if $\{(x_i, r_i)\}_{i\in\mathcal{C}}$ completely explains the corrupted parties' behavior. If the challenger generates $\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}$ using the honest protocol, then the behavior of $\mathcal{A}'$ is the same as the Hyb_7 experiment. If the challenger generates $\{\mathsf{msg}^{\mathsf{out}}_{i,r}\}_{i\in[M]\setminus\mathcal{C}}$ using the P2P semi-malicious simulator, then the behavior of $\mathcal{A}'$ is the same as the Hyb_8 experiment. Thus, since $\mathcal{A}$ successfully distinguishes between the two experiments with non-negligible probability, $\mathcal{A}'$ successfully wins the P2P semi-malicious simulator indistinguishability game with non-negligible probability. □

8.5 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" compilerfrom Appendix A. Then, we can compile it into a maliciously secure protocol with our "P2P semi-malicious to malicious secure" compiler from Section 8. 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 7. This results with a protocol in the random oracle model (which is somewhat inherent due to our lower bound from Lemma 5.1). 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} (see Step 4.f), 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 (see Step 4.h) . 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 Step 4.f 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 (see Step 9 in the evaluation phase), 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. Ilan Komargodski is the incumbent of the Harry & Abe Sherman Senior Lectureship at the School of Computer Science and Engineering at the Hebrew University. 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

[ABB+17] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet edcs: algorithms for matching and vertex cover on massive graphs. *arXiv preprint arXiv:1711.03076*, 2017.

[AG18] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *TOPC*, 2018.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.

[AK17] Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *SPAA*, 2017.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *STOC*, 2014.

[Ass17] Sepehr Assadi. Simple round compression for parallel vertex cover. *CoRR*, abs/1709.04599, 2017.

[ASW18] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. *CoRR*, abs/1805.02974, 2018.

[ASZ19] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log diameter rounds algorithms for 2-vertex and 2-edge connectivity. In *ICALP*, 2019.

[BBD$^+$19] Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively parallel computation of matching and MIS in sparse graphs. In *PODC*, 2019.

[BBLM14] MohammadHossein Bateni, Aditya Bhaskara, Silvio Lattanzi, and Vahab Mirrokni. Distributed balanced clustering via mapping coresets. In *in NeurIPS*, 2014.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, 2013.

[BCL$^+$21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In *Advances in Cryptology - CRYPTO*, pages 681–710, 2021.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *Theory of Cryptography - TCC*, pages 1–18, 2020.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *CRYPTO*, 2015.

[BCTV17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[BGG$^+$18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO*, pages 565–596, 2018.

[BGI$^+$12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 2012.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 501–519, 2014.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, 1988.

[BHH19] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *FOCS*, 2019.

[BJMS18] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Threshold multi-key fhe and applications to round-optimal mpc. *IACR Cryptology ePrint Archive*, page 580, 2018.

[BJPY18] Elette Boyle, Abhishek Jain, Manoj Prabhakaran, and Ching-Hua Yu. The bottleneck complexity of secure multiparty computation. In *ICALP*, 2018.

[BKV12] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and mapreduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

[BMV$^+$12] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[BP16] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multi-key FHE with short ciphertexts. In *Advances in Cryptology - CRYPTO*, pages 190–213, 2016.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology - ASIACRYPT*, pages 280–300, 2013.

[CCLS20] T.-H. Hubert Chan, Kai-Min Chung, Wei-Kai Lin, and Elaine Shi. MPC for MPC: secure computation on a massively parallel computing architecture. In *ITCS*, 2020.

[CFG$^+$19] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of $(\Delta{+}1)$ coloring in congested clique, massively parallel computation, and centralized local computation. In *PODC*, 2019.

[CŁM$^+$18] Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *STOC*, 2018.

[CM15] Michael Clear and Ciaran McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In *Advances in Cryptology - CRYPTO*, pages 630–656, 2015.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Innovations in Computer Science - ICS*, pages 310–331, 2010.

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster computing in zero knowledge. In *Advances in Cryptology - EUROCRYPT*, pages 371–403, 2015.

[DHRW16] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO*, 2016.

[dPBENW16] Rafael da Ponte Barbosa, Alina Ene, Huy L Nguyen, and Justin Ward. A new framework for distributed submodular maximization. In *FOCS*, pages 645–654, 2016.

[EIM11] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *SIGKDD*, 2011.

[EN15] Alina Ene and Huy Nguyen. Random coordinate descent methods for minimizing decomposable submodular functions. In *ICML*, 2015.

[FKLS20] Rex Fernando, Ilan Komargodski, Yanyi Liu, and Elaine Shi. Secure massively parallel computation for dishonest majority, 2020.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing, STOC*, pages 169–178, 2009.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *FOCS*, pages 464–479, 1984.

[GKMS19] Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *PODC*, 2019.

[GLM19] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. Improved parallel algorithms for density-based network clustering. In *ICML*, 2019.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2*. Cambridge university press, 2009.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[GU19] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *SODA*, 2019.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *STOC*, 2011.

[HSS19] MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively parallel approximation algorithms for edit distance and longest common subsequence. In *SODA*, 2019.

[HW15] Pavel Hubáček and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *ITCS*, pages 163–172, 2015.

[KMVV15] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *TOPC*, 2(3):14:1–14:22, 2015.

[KOS03] Jonathan Katz, Rafail Ostrovsky, and Adam D. Smith. Round efficiency of multi-party computation with a dishonest majority. In *Eurocrypt*, 2003.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, pages 669–684, 2013.

[KSV10] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, 2011.

[ŁMW18] Jakub Łącki, Vahab S. Mirrokni, and Michal Wlodarczyk. Connected components at scale via local contractions. *CoRR*, abs/1807.10727, 2018.

[LNO13] Yehuda Lindell, Kobbi Nissim, and Claudio Orlandi. Hiding the input-size in secure two-party computation. In *Advances in Cryptology - ASIACRYPT*, pages 421–440, 2013.

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, 2012.

[Mic94] Silvio Micali. CS proofs (extended abstracts). In *FOCS*, 1994.

[MKSK13] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NeurIPS*, 2013.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Eurocrypt*, 2016.

[NN01] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

[Ona18] Krzysztof Onak. Round compression for parallel graph algorithms in strongly sublinear space. *CoRR*, abs/1807.08745, 2018.

[Pas04] Rafael Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. In László Babai, editor, *STOC*, 2004.

[PS16] Chris Peikert and Sina Shiehian. Multi-key FHE from LWE, revisited. In *TCC*, 2016.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[RMCS13] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *ICDE*, 2013.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, 2014.

[Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *FOCS*, pages 80–91. IEEE Computer Society, 1982.

[YV18] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. In *ICML*, 2018.

Supplementary Material

A Semi-Malicious Secure MPC for Short Output

In this section, we give a semi-malicious compiler for MPC protocols whose output is short enough to fit into the memory of a single machine. The compiler takes as input an arbitrary "short output" (posssibly insecure) MPC protocol and transforms it into a semi-malicious counterpart. The compiler is basically the same as the one of Fernando et al. [FKLS20], but they only proved that it is secure in the semi-honest setting. We extend their proof of security to show that the compiler is readily semi-malicious.

Theorem A.1 (Semi-Malicious Secure MPC for Short 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, and assuming Π results in an output for party 1 and no output for any other party. Assume that $M \in \mathsf{poly}(\lambda)$ and $\lambda \leq S$. Further, assume that there is a (non-leveled) threshold FHE scheme as in Section 3.4.*

Then, there is an algorithm (compiler) that transforms Π into another protocol $\tilde{\Pi}$ which assumes a PKI 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.*

A.1 The Protocol

As mentioned the protocol that satisfies the properties in Theorem A.1 is the same as that of Fernando et al. [FKLS20] and we give it below for completeness. We note that the proof of semi-malicious security is slightly different from the proof of semi-honest security; details below.

The protocol proceeds in a encrypt-evaluate-decrypt fashion, where the encryption and evaluation are done using a (threshold) FHE scheme and the decryption is done by aggregating all partial decryptions in a tree. In more detail, the protocol proceeds in two phases: first, each party encrypts its initial state under the public key pk, and the parties carry out an encrypted version of the original (insecure) MPC protocol using the TFHE evaluation function. Second, P_1 distributes the resulting ciphertext, which is an encryption of the output, and all parties compute and combine their partial decryptions so that P_1 learns the decrypted output. This second phase crucially relies on the fact that the TFHE scheme partial decryptions can be combined in a tree-like fashion.

The formal description of the protocol is below. Note that we use two subprotocols $\mathsf{Distribute}$ and $\mathsf{Combine}$, which are given in Section 6.

P2P Semi-Malicious Compiler for Short-output Protocols

Input: Party P_i has input x_i to the underlying MPC protocol and circuits $\mathsf{NextSt}_{i,1}, \ldots, \mathsf{NextSt}_{i,R}$, as described in Section 4.1.

Output: In the end of the protocol, party P_1 will receive the output $y \in \{0,1\}^{l_{\mathsf{out}}}$, where $y = f(x_1, \ldots, x_M)$ and f is the functionality that the original protocol Π computes.

1: **Setup Phase:** Each party P_i knows the security parameter λ, the public key pk along with a secret key sk_i, where $(pk, sk_1, \ldots, sk_M) \leftarrow \mathsf{TFHE.Setup}(1^\lambda, M)$.

2: **Evaluation Phase:**

(a) Each party P_i encrypts its input x_i using the public key pk: $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}(\mathsf{pk}, x_i)$.

(b) Parties evaluate the underlying protocol round by round. During round $r \in [R]$, each party P_i computes $ct_{\mathsf{st}_{i,r}} || ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}} \leftarrow \mathsf{TFHE.Eval}(\mathsf{NextSt}_{i,r}, ct_{\mathsf{st}_{i,r-1}} || ct_{\mathsf{msg}^{\mathsf{in}}_{i,r-1}})$ and sends encrypted messages $ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}}$ to other parties according to the original protocol Π.

(c) In the end of the evaluation phase, party P_1 holds ct_{out}, the encryption of the output y.

3: **Decryption Phase:**

(a) Parties execute $\mathsf{Distribute}_\lambda(ct_{\mathsf{out}})$ so that all parties receive the encryption ct_{out}.

(b) Each party P_i locally computes $\rho_i \leftarrow \mathsf{TFHE.PartDec}(\mathsf{sk}_i, ct_{\mathsf{out}})$.

(c) Parties run $\mathsf{Combine}_\lambda(+, \{\rho_i\}_{i\in[M]})$ and party P_1 receives $\rho = \sum_{i=1}^{M} \rho_i$.

4: **Offline Output Decryption Phase:** Once P_1 knows ρ he can obtain the output $y \leftarrow \mathsf{TFHE.Dec.Round}(\rho)$.

Correctness and efficiency. Correctness follows directly from correctness of the TFHE scheme. We proceed with the efficiency analysis. Note that, since $M \in \mathsf{poly}(\lambda)$ that mean there exists some constant $\epsilon > 0$, such that $\lambda \le M^\epsilon$, so $\lceil \log_\lambda M \rceil \le \lceil \log_{M^\epsilon} M \rceil = \lceil \epsilon^{-1} \rceil = O(1)$.

The encrypted MPC phase takes exactly R rounds, the distributed output decryption phase consists of $\mathsf{Distribute}$ and $\mathsf{Combine}$, both of which take $\lceil \log_\lambda M \rceil = O(1)$ rounds. Each party P_i in the evaluation phase uses $S \cdot \mathsf{poly}(\lambda)$ space. During decryption phase a $\mathsf{Distribute}$ and $\mathsf{Combine}$ sub-protocols are invoked. There, P_i uses $O(l_{out} \cdot \lambda^2)$ space, since it stores λ ciphertexts, each of size $l_{out} \cdot \lambda$. So the total space complexity is also $S \cdot \mathsf{poly}(\lambda)$.

A.2 Security

To prove security, for every P2P semi-malicious adversary, we exhibit a simulator for the protocol given above. This simulator will generate a view of an arbitrary set of corrupted parties which will be indistinguishable from the view of the corrupted parties in a real-world execution of the protocol. Note that the simulator receives the public key which is assumed to be generated honestly by the TFHE setup algorithm, and also receives the set of corrupted parties $\mathcal{C}$ as input. This allows the corrupted set $\mathcal{C}$ to be chosen based on the public key.

The behavior of the simulator is described below.

Short Output Simulator

Input: The simulator receives the corrupted set $\mathcal{C}$, the public key pk, the corrupted parties' inputs $\{x_i\}_{i\in\mathcal{C}}$, and, if $1 \in \mathcal{C}$, the output $y = f(x_1, \ldots, x_M)$.

Output: In the end of the protocol, if $1 \in \mathcal{C}$, party P_1 will receive the output $y = f(x_1, \ldots, x_M)$ and f is the functionality that the original protocol Π computes.

1: **Simulated Setup:** To generate the corrupted parties' secret keys, the simulator uses the TFHE simulated setup: $(\{sk_i\}_{i\in\mathcal{C}}, \sigma_{sim}) \leftarrow \mathsf{TFHE.Sim.Setup}(pk, \mathcal{C})$, and sends $\{sk_i\}_{i\in\mathcal{C}}$ to the adversary.

2: **Simulated Evaluation Phase:**

(a) For each honest party $i \in [M] \setminus \mathcal{C}$, the simulator computes an encryption of 0: $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{|x_i|})$

(b) The simulator executes the underlying protocol round by round with the adversary. During round $r \in [R]$, each party P_i computes $ct_{\mathsf{st}_{i,r}} || ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}} \leftarrow \mathsf{TFHE.Eval}(\mathsf{NextSt}_{i,r}, ct_{\mathsf{st}_{i,r-1}} || ct_{\mathsf{msg}^{\mathsf{in}}_{i,r-1}})$ and sends encrypted messages $ct_{\mathsf{msg}^{\mathsf{out}}_{i,r}}$ to other parties according to the original protocol Π.

(c) In the end of the evaluation phase, party P_1 holds ct_{out}, the encryption of the output y.

3: **Simulated Decryption Phase:**

(a) The simulator invoke $\mathsf{Distribute}_\lambda(ct_{\mathsf{out}})$ with the adversary, so that all parties receive the encryption ct_{out}.

(b) The simulator use the inputs and the adversary witness tape to obtain $\{ct_{\mathsf{st}_{i,0}}\}_{i \in \mathcal{C}}$.

(c) The simulator invoke the TFHE simulator to obtain simulated partial decryptions: $\{\rho_i\}_{i \in [M] \setminus \mathcal{C}} \leftarrow \mathsf{TFHE.Sim.Query}(f, \{ct_{\mathsf{st}_{i,0}}\}_{i \in [M]}, y, [M] \setminus \mathcal{C}, \sigma_{sim})$, or if $1 \notin \mathcal{C}$, $\{\rho_i\}_{i \in [M] \setminus \mathcal{C}} \leftarrow \mathsf{TFHE.Sim.Query}(f, \{ct_{\mathsf{st}_{i,0}}\}_{i \in [M]}, \perp, [M] \setminus \mathcal{C}, \sigma_{sim})$.

(d) Parties run $\mathsf{Combine}_\lambda(+, \{\rho_i\}_{i \in [M]})$ and party P_1 receives $\sum_{i=1}^{M} \rho_i$.

We now prove indistinguishability of the simulated views in the short output protocol from the real-world views. That is, we show that no efficient adversary which corrupts an arbitrary set of parties can distinguish between the output of the simulator and the view of the corrupted parties in an honest execution of the protocol. We show this via a sequence of computationally indistinguishable hybrids, where in the first hybrid the output of the simulator corresponds to the real world, and in the last hybrid the simulator's output corresponds to the ideal world.

- **Hybrid H_0:** In this hybrid, the simulator behaves identically to the real world, setting the corrupted parties' secret keys to be the ones generated by the TFHE setup, and running the real-world protocol.

- **Hybrid H_1:** In this hybrid, the simulator behaves the same as in H_0, except it generates the corrupted parties' secret keys via $\mathsf{TFHE.Sim.Setup}$, and uses $\mathsf{TFHE.Sim.Query}$ to generate the honest parties' partial decryptions for the output. Note that if $1 \notin \mathcal{C}$ then the simulator passes $\perp$ to $\mathsf{TFHE.Sim.Query}$ instead of the output $f(x_1, \ldots, x_M)$.

- **Hybrid H_2:** In this hybrid, the simulator behaves the same as in H_1, except that it sets the encryptions of each honest party's input to 0: $ct_{\mathsf{st}_{i,0}} \leftarrow \mathsf{TFHE.Enc}_{pk}(0^{|x_i|})$. H_2 is identical to the ideal world.

We now show indistinguishability of each successive pair of hybrids, relying on the properties of the TFHE scheme which are defined in Section 3.4.

Claim 10. *Assuming simulation security of the TFHE scheme, and simulation of incomplete decryptions, the output of the simulator in H_0 is indistinguishable from the output of the simulator in H_1.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between H_0 and H_1. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against simulation security of the TFHE scheme.

$\mathcal{A}'$ performs the same steps as the simulator in H_0, except that it interacts with the challenger to obtain the public key, the corrupted parties' secret keys, the initial ciphertexts and the partial decryptions of the honest parties. It first receives pk from the challenger, then sends this to $\mathcal{A}$, receives $\mathcal{C}$ from $\mathcal{A}$, along with the plaintexts $\{x_i\}_{i\in[M]}$, where x_i is P_i's input for the underlying MPC protocol, and sends this to the challenger. When it receives back ciphertexts $\{ct_i\}_{i\in[M]\setminus\mathcal{C}}$, and sends this to $\mathcal{A}$, receives $\{ct_i\}_{i\in\mathcal{C}}$ from $\mathcal{A}$, it then performs the encrypted MPC phase in the same way as in H_0, and then finally, for the distributed output computation phase, it sends the query (I, f), where f is the circuit representing the underlying MPC protocol and I is the set of honest parties whose partial decryptions are seen by parties in $\mathcal{C}$. Note that if $I \cup \mathcal{C} \neq [M]$, which happens when $1 \notin \mathcal{C}$, then the challenger uses $\perp$ instead of $f(x_1, \ldots, x_M)$ when evaluating $\mathsf{TFHE.Sim.Query}$. When $\mathcal{A}'$ receives the partial decryptions of the honest parties, it then uses these partial decryptions to perform the rest of the procedure in the same way as H_0. Once it has finished, it sends the views of the corrupted parties, and sends this view to $\mathcal{A}$. It outputs the output of $\mathcal{A}$.

If the TFHE simulation security challenger enacts the real-world experiment, then the view of $\mathcal{A}$ is identical to H_0. If the TFHE challenger enacts the ideal-world experiment then the view of $\mathcal{A}$ is identical to H_1. Thus if $\mathcal{A}$ distinguishes between the hybrids then $\mathcal{A}'$ is an efficient distinguisher between the real and ideal experiments of the TFHE simulation game. □

Claim 11. *Assuming semantic security of the TFHE scheme, the output of the simulator in H_1 is indistinguishable from the output of the simulator in H_2.*

Proof. Assume that there is an efficient adversary $\mathcal{A}$ which distinguishes between H_1 and H_2 with non-negligible probability. We use $\mathcal{A}$ to build a reduction $\mathcal{A}'$ against semantic security of the TFHE scheme.

$\mathcal{A}'$ sends the set $\mathcal{C}$ of corrupted parties to the TFHE challenger. It receives pk and the secret keys sk_i for $i \in \mathcal{C}$. It then performs the same steps as the simulator in H_1, except that it sends the honest parties' inputs to the challenger, and uses the ciphertexts which the challenger provides as the honest parties' encrypted initial state. It sends the view of the corrupted parties to $\mathcal{A}$, and outputs the output of $\mathcal{A}$.

If the TFHE semantic security challenger sends ciphertexts with the true values, then the view of $\mathcal{A}$ is identical to H_1. If the TFHE challenger sends encryptions of 0 then the view of $\mathcal{A}$ is identical to H_2. Thus, if $\mathcal{A}$ can distinguish between the hybrids then $\mathcal{A}'$ is a successful efficient adversary against the TFHE semantic security game. □