Order-C Secure Multiparty Computation for Highly Repetitive Circuits

Gabrielle Beck[1], Aarushi Goel[1], Abhishek Jain[1], and Gabriel Kaptchuk[2]

[1] Johns Hopkins University {becgabri,aarushig,abhishek}@cs.jhu.edu
[2] Boston University kaptchuk@bu.edu

Abstract. Running secure multiparty computation (MPC) protocols with hundreds or thousands of players would allow leveraging large volunteer networks (such as blockchains and Tor) and help justify honest majority assumptions. However, most existing protocols have at least a linear (multiplicative) dependence on the number of players, making scaling difficult. Known protocols with asymptotic efficiency independent of the number of parties (excluding additive factors) require expensive circuit transformations that induce large overheads.

We observe that the circuits used in many important applications of MPC such as training algorithms used to create machine learning models have a *highly repetitive structure*. We formalize this class of circuits and propose an MPC protocol that achieves $O(|\mathsf{C}|)$ total complexity for this class. We implement our protocol and show that it is practical and outperforms $O(n|\mathsf{C}|)$ protocols for modest numbers of players.

1 Introduction

Secure Multiparty Computation (MPC) [Yao86,GMW87,CCD88,BGW88] is a technique that allows mutually distrusting parties to compute an arbitrary function without revealing anything about the parties' private inputs, beyond what is revealed by the function output. In this work, we focus on *honest-majority* MPC, where a majority of the participants are assumed to be honest.

As public concern over privacy and data sharing grows, MPC's promise of privacy preserving collaboration becomes increasingly important. In recent years, MPC techniques are being applied to an increasingly complex class of functionalities such as distributed training of machine learning networks. Most current applications of MPC, however, focus on using a *small* number of parties. This is largely because most known (and all implemented) protocols incur a linear multiplicative overhead in the number of players in the communication and computation complexity, *i.e.* have complexity $O(n|\mathsf{C}|)$[3], where n is the number of players and $|\mathsf{C}|$ is the size of the circuit [HN06,DN07,LN17,CGH+18,NV18,FL19].

The Need for Large-Scale MPC. Yet, the most exciting MPC applications are at their best when a *large* number of players can participate in the protocol. These include crowd-sourced machine learning and large scale data collection, where widespread participation would result in richer data sets and more robust conclusions. Moreover, when the number of participating players is large, the honest majority assumption – which allows for the most efficient known protocols till date – becomes significantly more believable. Indeed, the honest majority of resources assumptions (a different but closely related set of assumptions) in Bitcoin [Nak08] and TOR [RSG98,DMS04] appear to hold up in practice when there are many protocol participants.

Furthermore, large-scale volunteer networks have recently emerged, like Bitcoin and TOR, that regularly perform incredibly large distributed computations. In the case of cryptocurrencies, it would be beneficial to apply the computational power to more interesting applications than mining, including executions of MPC protocols. Replicating a fraction of the success of these networks could enable massive, crowd-sourced applications that still respect privacy. In fact, attempts to run MPC on such large networks have already started [WJS+19], enabling novel measurements.

[3] For sake of simplicity, throughout the introduction, we omit a linear multiplicative factor of the security parameter in all asymptotic notations.

Our Goal: Order-C MPC. It would be highly advantageous to go beyond the limitation of current protocols and have access to an MPC protocol with total computational and communication complexity $O(|\mathsf{C}|)$.

Such a protocol can support division of the total computation among players which means that using large numbers of players can significantly reduce the burden on each individual participant. In particular, when considering complex functions, with circuit representations containing tens or hundreds of millions of gates, decreasing the workload of each individual party can have a significant impact. Ideally, it would be possible for the data providers themselves, possibly using low power or bandwidth devices, to participate in the computation.

An $O(|\mathsf{C}|)$ MPC protocol can also offer benefits in the design of other cryptographic protocols. In [IKOS07], Ishai et al. showed that zero-knowledge (ZK) proofs [GMR85] can be constructed using an "MPC-in-the-head" approach, where the prover simulates an MPC protocol in their mind and the verifier selects a subset of the players views to check for correctness. The efficiency of these proofs is inherited from the complexity of the underlying MPC protocols, and the soundness error is a function of the number of views opened and the number of players for which a malicious prover must have to "cheat" in order to control the protocol's output. This creates a tension: higher number of players can be used to increase the soundness of the ZK proof, but simulating additional players increases the complexity of the protocol. Access to an $O(|\mathsf{C}|)$ MPC protocol would ease this tension, as a large numbers of players could be used to simulate the MPC without incurring additional cost.

Despite numerous motivations and significant effort, there are no known $O(|\mathsf{C}|)$ MPC protocols for "non-SIMD" functionalities.[4] We therefore ask the following:

Is it possible to design an MPC protocol with $O(|\mathsf{C}|)$ total computation (supporting division of labor) and $O(|\mathsf{C}|)$ total communication?

Prior Work: Achieving $\tilde{O}(|\mathsf{C}|)$-MPC. A significant amount of effort has been devoted towards reducing the asymptotic complexity of (honest-majority) MPC protocols, since the initial $O(n^2|\mathsf{C}|)$ protocols [BGW88,CCD88].

Over the years, two primary techniques have been developed for reducing protocol complexity. The first is an *efficient multiplication protocol* combined with batched correlated randomness generation introduced in [DN07]. Using this multiplication protocol reduces the (amortized) complexity of a multiplication gate from $O(n^2)$ to $O(n)$, effectively shaving a factor of n from the protocol complexity. The second technique is *packed secret sharing* (PSS) [FY92], a vectorized, single-instruction-multiple-data (SIMD) version of traditional threshold secret sharing. By packing $\Theta(n)$ elements into a single vector, $\Theta(n)$ operations can be performed at once, reducing the protocol complexity by a factor of n when the circuit structure is accommodating to SIMD operations. Using these techniques separately, $O(n|\mathsf{C}|)$ protocols were constructed in [DI06] and [DN07].

While it might seem as though combining these two techniques would result in an $O(|\mathsf{C}|)$ protocol, the structural requirements of SIMD operations make it unclear on how to do so. The works of [DIK+08] and [DIK10] demonstrate two different approaches to combine these techniques, either by relying on randomizing polynomials or using circuit transformations that involve embedding routing networks within the circuits. These approaches yield $\widetilde{O}(|\mathsf{C}|)$ protocols with large multiplicative constants and additive terms that depend on the circuit depth. (The additive terms were further reduced in the recent work of [GIP15].)

In summary, while both PSS and efficient multiplication techniques have been known for over a decade, no $O(|\mathsf{C}|)$ MPC protocols are known. The best known asymptotic efficiency is $\widetilde{O}(|\mathsf{C}|)$ achieved by [DIK+08,DIK10,GIP15]; however, these protocols have never been implemented for reasons discussed above. Instead, the state-of-the-art implemented protocols achieve $O(n|\mathsf{C}|)$ computational and communication efficiency [CGH+18,NV18,FL19].

[4] SIMD circuits are arithmetic circuits that simultaneously evaluate ℓ copies of the same arithmetic circuit on different inputs. Genkin et al. [GIP15] showed that it is possible to design an $O(|\mathsf{C}|)$ MPC protocol for SIMD circuits, where $\ell = \Theta(n)$.

1.1 Our Contributions

In this work, we identify a meaningful class of circuits, called (A, B)-repetitive circuits, parameterized by variables A and B. We show that for $(\Omega(n), \Omega(n))$-repetitive circuits, efficient multiplication and PSS techniques can indeed be combined, using new ideas, to achieve $O(|\mathsf{C}|)$ MPC for n parties. To the best of our knowledge, this is the first such construction for a larger class of circuits than SIMD circuits.

We test the practical efficiency of our protocol by means of a preliminary implementation and show via experimental results that for computations involving large number of parties, our protocol outperforms the state-of-the-art implemented MPC protocols. We now discuss our contributions in more detail.

Highly Repetitive Circuits. The class of (A, B)-repetitive circuits are circuits that are composed of an arbitrary number of *blocks* (sets of gates at the same depth) of width at least A, that recur at least B times throughout the circuit. Loosely speaking, we say that an (A, B)-repetitive circuit is *highly repetitive* w.r.t. n parties, if $A \in \Omega(n)$ and $B \in \Omega(n)$.

The most obvious example of this class includes the sequential composition of some (possibly multi-layer) functionality, i.e. $f(f(f(f(\ldots))))$ for some arbitrary f with sufficient width. However, this class also includes many other types of circuits and important functionalities. For example, as we discuss in Section 4.3, machine learning model training algorithms (many iterations of gradient descent) are highly repetitive even for large numbers of parties. We also identify block ciphers and collision resistant hash functions as having many iterated rounds; as such functions are likely to be run many times in a large-scale, private computation, they naturally result in highly repetitive circuits for larger numbers of parties. We give formal definition of (A, B)-repetitive circuits in Section 4.

Semi-Honest Order-C MPC. Our primary contribution is a semi-honest, honest-majority MPC protocol for highly repetitive circuits with *total computation and communication complexity* $O(|\mathsf{C}|)$. Our protocol only requires communication over point-to-point channels and works in the plain model (i.e., without trusted setup). It achieves unconditional security against $t < n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$ corruptions, where ϵ is a tunable parameter as in prior works based on PSS.

Our key insight is that the repetitive nature of the circuit can be leveraged to efficiently generate correlated randomness in a way that helps overcome the limitations of PSS. We elaborate on our techniques in Section 2.

Malicious Security Compiler. We next consider the case of malicious adversaries. In recent years, significant work [GIP+14,GIP15,LN17,CGH+18,NV18,FL19,GSZ20] has been done on designing efficient malicious security compilers for honest majority MPC. With the exception of [GIP15], all of these works design compilers for protocols based on regular secret sharing (SS) as opposed to PSS. The most recent of these works [CGH+18,NV18,FL19,GSZ20] achieve very small constant multiplicative overhead, and ideally one would like to achieve similar efficiency in the case of PSS-based protocols.

Since our semi-honest protocol is based on PSS, the compilers of [CGH+18,NV18,FL19,GSZ20] are not directly applicable to our protocols. Nevertheless, borrowing upon the insights from [GIP15], we demonstrate that the techniques developed in [CGH+18] can in fact be used to design an efficient malicious security compiler for our PSS-based semi-honest protocol. Specifically, our compiler incurs a multiplicative overhead of approximately 1.6–2.3, depending on the choice of ϵ, over our semi-honest protocol for circuits over large fields (where the field size is exponential in the security parameter).[5] For circuits over smaller fields, the multiplicative overhead incurred is $O(k/\log|\mathbb{F}|)$, where k is the security parameter and $|\mathbb{F}|$ is the field size.

Efficiency. We demonstrate that our protocol is not merely of theoretical interest but is also concretely efficient for various choices of parameters. We give a detailed complexity calculation of our protocols in Sections 7 and 8.5.

For $n = 125$ parties and $t < n/3$, our malicious secure protocol only requires each party to, on average, communicate approximately $2\frac{3}{4}$ *field elements per gate* of a highly repetitive circuit. In contrast, the

[5] We note that for more commonly used corruption thresholds $n/2 > t > n/4$, the overhead incurred by our compiler is approximately 2.3.

state-of-the-art [FL19] (an information-theoretic $O(n|\mathsf{C}|)$ protocol for $t < n/3$) requires each party to communicate approximately $4\frac{2}{3}$ field elements per multiplication gate. Thus, (in theory) we expect our protocol to outperform [FL19] for circuits with around 65% multiplication gates with just 125 parties. Since the per-party communication in our protocol decreases as the number of parties increase, our protocol is expected to perform better as the number of parties increase.

We confirm our conjecture via a preliminary implementation of our malicious secure protocol and give concrete measurements of running it for up to 300 parties, across multiple network settings. Since state-of-the-art honest-majority MPC protocol have only been tested with smaller numbers of parties, we show that our protocol is comparably efficient even for fewer number of parties. Moreover, our numbers suggest that our protocol would outperform these existing protocols when executed with hundreds or thousands of players at equivalent circuit depths.

Application to Zero-Knowledge Proofs. The *MPC-in-the-head* paradigm of Ishai et al. [IKOS07] is a well-known technique for constructing efficient three-round public-coin honest-verifier zero-knowledge proof systems (aka sigma protocols) from (honest-majority) MPC protocols. Such proof systems can be made *non-interactive*, in the random oracle model [BR93] via the Fiat-Shamir paradigm [FS87]. Recent works have demonstrated the practical viability of this approach by constructing zero-knowledge proofs [GMO16,CDG+17,KKW18,AHIV17] where the proof size has linear or sub-linear dependence on the size of the relation circuit.

Our malicious-secure MPC protocol can be used to instantiate the MPC-in-the-head paradigm when the relation circuit has highly repetitive form. The size of the resulting proofs will be comparable to the best-known *linear-sized* proof system constructed via this approach [KKW18]. Importantly, however, it can have more efficient prover and verifier computation time. This is because [KKW18] requires parallel repetition to get negligible soundness, and have computation time linear in the number of simulated players. Our protocol (by virtue of being an Order-C and honest majority protocol), on the other hand, can accommodate massive numbers of (simulated) parties without increasing the protocol simulation time and achieve small soundness error without requiring additional parallel repetition. Finally, we note that sublinear-sized proofs [AHIV17] typically require super-linear prover time, in which case simulating our protocol may be more computationally efficient for the prover. We leave further exploration of this direction for future work.

Concurrent Work and Future Directions. Our protocols achieve $O(|\mathsf{C}|)$ complexity for a large class of non-SIMD circuits, namely, highly repetitive circuits. A natural open question is whether it is possible to extend our work to handle other classes of circuits.

Concurrent to our work, [GSY21] also study the problem of scalable MPC protocols. They make use of packed secret sharing to generate Beaver triples for every multiplication gate in the pre-processing phase with $O(|\mathsf{C}|)$ total communication. In the online phase, the parties "unpack" these shares and use them individually for each multiplication. While a naive implementation of the online phase would require $O(n|\mathsf{C}|)$ communication, their protocol reduces communication by electing small committees who run different parts of the online phase. However, this approach falls short of achieving $O(|\mathsf{C}|)$ communication in the online phase since the number and sizes of the committees are not constant. Unlike our work, they consider general circuits but rely on computation assumptions.

Another important direction for future work is to further improve upon the concrete efficiency of our semi-honest $O(|\mathsf{C}|)$ protocol. The multiplicative constant in our protocol complexity is primarily dictated by the tunable parameter ϵ, which is inherent in PSS-based protocols. Thus, achieving improvements on this front will likely require different techniques.

Our malicious security compiler, which builds on ideas from [CGH+18], incurs a multiplicative overhead of of approximately 2.3, over the semi-honest protocol. Recent works of [FL19,GSZ20] achieve even lower overheads than the compiler of [CGH+18]. Another useful direction would be to integrate our ideas with the techniques in [FL19,GSZ20] (possibly for a lower corruption threshold) to obtain more efficient compilers for PSS-based protocols. We leave this for future work.

2 Technical Overview

We begin our technical overview by recalling the key techniques developed in prior works for reducing dependence on the number of parties. We then proceed to describe our main ideas in Section 2.2.

2.1 Background

Classical MPC protocols have communication and computation complexity $O(n^2|\mathsf{C}|)$. These protocols, exemplified by [BGW88], leverage Shamir's secret sharing [Sha79a] to facilitate distributed computation and require communication for each multiplication gate to enable degree reduction. Typical multiplication subprotocols require that each party send a message to every other party for every multiplication gate, resulting in total communication complexity $O(n^2|\mathsf{C}|)$. As mentioned earlier, two different techniques have been developed to reduce the asymptotic complexity of MPC protocols down to $O(n|\mathsf{C}|)$: efficient multiplication techniques and packed secret sharing.

Efficient Multiplication. In [DN07], Damgård and Nielsen develop a randomness generation technique that allows for a more efficient multiplication subprotocol. At the beginning of the protocol, the parties generate shares of random values, planning to use one of these values for each multiplication gate. These shares are generated in *batches*, using a subprotocol requiring $O(n^2)$ communication that outputs $\Theta(n)$ shares of random values. This *batched randomness generation* subprotocol can be used to compute $O(|\mathsf{C}|)$ shared values with total complexity $O(n|\mathsf{C}|)$. After locally evaluating a multiplication gate, the players use one of these shared random values to mask the gate output. Players then send the masked gate output to a *leader*, who reconstructs and broadcasts the result back to all players.[6] Finally, players locally remove the mask to get a shared value of the appropriate degree. This multiplication subprotocol has complexity $O(n)$.

Packed Secret Sharing. In [FY92], Franklin and Yung proposed a vectorized version of Shamir secret sharing called *packed secret sharing* that trades a lower corruption threshold for more efficient representation of secrets. More specifically, their scheme allows a dealer to share a vector of $\Theta(n)$ secrets such that each of the n players still only hold a single field element. Importantly, the resulting shares preserve a SIMD version of the homomorphisms required to run MPC. Specifically, if $X = (x_1, x_2, x_3)$ and $Y = (y_1, y_2, y_3)$ are the vectors that are shared and added or multiplied, the result is a sharing of $X + Y = (x_1 + y_1, x_2 + y_2, x_3 + y_3)$ or $XY = (x_1y_1, x_2y_2, x_3y_3)$ respectively. Like traditional Shamir secret sharing, the degree of the polynomial corresponding to XY is twice that of original packed sharings of X and Y. This allows players to compute over $O(n)$ gates *simultaneously*, provided two properties are satisfied: (1) all of the gates *perform the same operation* and (2) the inputs to each gate *are in identical positions in the respective vectors*. In particular, it is not possible to compute x_1y_2 in the previous example, as x_1 and y_2 are not *aligned*. However, if the circuit has the correct structure, packed secret sharing reduces MPC complexity from $O(n^2|\mathsf{C}|)$ to $O(n|\mathsf{C}|)$.

2.2 Our Approach: Semi-Honest Security

A Strawman Protocol. A natural idea towards achieving $O(|\mathsf{C}|)$ MPC is to design a protocol that can take advantage of *both* efficient multiplications and packed secret sharing. As each technique asymptotically shaves off a factor of n, we can expect the resulting protocol to have complexity $O(|\mathsf{C}|)$. A naïve (strawman) protocol combining these techniques might proceed as follows:

— Players engage in a first phase to generate packed shares of random vectors using the batching technique discussed earlier. This subprotocol requires $O(n^2)$ messages to generate $\Theta(n)$ shares of packed random values, each containing $\Theta(n)$ elements. As we need a single random value per multiplication gate, $O(|\mathsf{C}|)$ total messages are sent.

— During the input sharing phase, players generate packed shares of their inputs, distributing shares to all players.

[6] The choice of the leader can be rotated amongst the players to divide the total computation.

— Players proceed to evaluate the circuit over these packed shares, using a single leader to run the efficient multiplication protocol to reduce the degrees of sharings after multiplication. This multiplication subprotocol requires $O(n)$ communication to evaluate $\Theta(n)$ gates, so the total complexity is $O(|\mathsf{C}|)$.
— Once the outputs have been computed, players broadcast their output shares and reconstruct the output.

While natural, this template falls short because the circuit may not satisfy the requirements to perform SIMD computation over packed shares. As mentioned before, packed secret sharing only offers savings if all the simultaneously evaluated gates are the same and all gate inputs are properly aligned. However, this is an unreasonable restriction to impose on the circuits. Indeed, running into this problem, [DIK10,GIP15] show that any circuit can be modified to overcome these limitations, at the cost of a significant blowup in the circuit size, which adversely affects their computation and communication efficiency. (We discuss their approach in more detail later in this section.)

Our Ideas. Without such a circuit transformation, however, it is not immediately clear how to take advantage of packed secret sharing (other than for SIMD circuits). To address this challenge, we devise two conceptual tools, each of which we will "simulate" using existing primitives, as described below:

1. *Differing-operation packed secret sharing*, a variant of packet secret sharing in which different operations can be evaluated for each position in the vector. For example, players holding shares of (x_1, x_2, x_3) and (y_1, y_2, y_3) are unable to compute $(x_1y_1, x_2 + y_2, x_3y_3)$. With *differing-operation packed secret sharing*, we imagine the players can generate an operation vector (*e.g.* $(\times, +, \times)$) and apply the corresponding operation to each pair of inputs. Given such a primitive, there would be no need to modify a circuit to ensure that shares are evaluated on the same kind of gate.
2. *A realignment procedure* that allows pre-existing packed secret shares to be modified so previously unaligned vector entries can be moved and aligned properly for continued computation without requiring circuit modification.

We note that highly repetitive circuits are *layered* circuits (that is the inputs to layer $i + 1$ of a circuit are all output wires from layer i). For the remainder of this section, we will make the simplifying assumption that circuits contain only multiplication and addition gates and that the circuit is layered. We expand our analysis to cover other gates (e.g. relay gates) in the technical sections.

Simulating Differing-operation Packed Secret Sharing. To realize differing-operation packed secret sharing, we require the parties to compute *both* operations over their input vectors. For instance, if the player hold share of (x_1, x_2, x_3) and (y_1, y_2, y_3) and wish to compute the operation vector $(\times, +, \times)$, they begin by computing both $(x_1 + y_1, x_2 + y_2, x_3 + y_3)$ and (x_1y_1, x_2y_2, x_3y_3). Note that all the entries required for the final result are contained in these vectors, and the players just need to "select" which of the aligned entries will be included in the final result.

Recall that in the multiplication procedure described earlier, the leader reconstructs all masked outputs before resharing them. We modify this procedure to have the leader reconstruct both the sum and product of the input vectors, *i.e.* the unpacked values $x_1+y_1, x_2+y_2, x_3+y_3, x_1y_1, x_2y_2, x_3y_3$ (while masked). The leader then performs this "selection" process, and packs only the required values to get a vector $(x_1y_1, x_2+y_2, x_3y_3)$, and discards the unused values x_1+y_1, x_2y_2, x_3+y_3. Shares of this vector are then distributed to the rest of the players, who unmask their shares. Note that this procedure only has an overhead of 2, as both multiplication and addition must be computed.[7]

Simulating the Realignment Procedure. First note that realigning packed shares may require not only internal permutations of the shares, but also swapping values across vectors. For example, consider the circuit snippet depicted in Figure 1. The outputs of the green (bottom) layer are not structured correctly to enable computing the purple (top) layer, and require this cross-vector swapping. As such, we require a realignment procedure that takes in all the vectors output by computing a particular circuit layer and outputs multiple properly aligned vectors.

[7] In this toy example only one vector is distributed back to the parties. If layers are approximately of the same size, an approximately equal number of vectors will be returned.

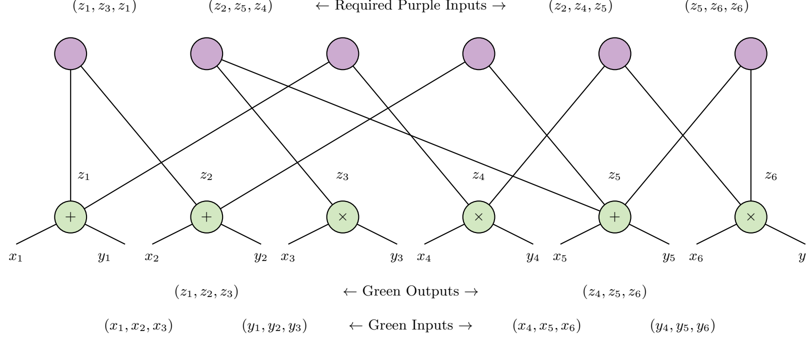


Fig. 1: A simple example pair of circuit layers illustrating the need for differing-operation packed secret sharing and our realignment procedure. Players begin by evaluating both addition and multiplication on each pair of input vectors. However, the resulting vectors are not properly aligned to compute the purple layer. To get properly aligned packings, the vectors $(z_1^{\text{add}}, z_2^{\text{add}}, z_3^{\text{add}}), (z_1^{\text{mult}}, z_2^{\text{mult}}, z_3^{\text{mult}})$ and $(z_4^{\text{add}}, z_5^{\text{add}}, z_6^{\text{add}}), (z_4^{\text{mult}}, z_5^{\text{mult}}, z_6^{\text{mult}})$ are masked and opened to the leader. The leader repacks these values such that the resulting vectors will be properly aligned for computing the purple layer. For instance, in this case the leader would deal shares of $(z_1^{\text{add}}, z_3^{mult}, z_1^{\text{add}}), (z_2^{add}, z_5^{\text{add}}, z_4^{mult}), (z_2^{\text{add}}, z_4^{mult}, z_5^{\text{add}}),$ and $(z_5^{\text{add}}, z_6^{mult}, z_6^{\text{mult}})$

Our realignment procedure builds on the ideas used to realize differing-operation packed secret sharing. Recall that the leader is responsible for reconstructing the masked result values from *all* gates in the previous layer. With access to all these masked values, the leader is not only able to select between a pair of values for each element of a vector (as before), but instead can arbitrarily select the values required from across all outputs. For instance, in the circuit snippet in Figure 1, the leader has masked, reconstructed values $z_i^{add}, z_i^{\text{mult}}$ for $i \in [6]$. Proceeding from left to right of the purple layer, the leader puts the value corresponding to the left input wire of a gate into a vector and the right input wire value into the correctly aligned slot of a corresponding vector. Using this procedure, the input vectors for the first three gates of the purple layer will be $(z_1^{add}, z_3^{\text{mult}}, z_1^{\text{add}})$ (left wires) and $(z_2^{add}, z_5^{add}, z_4^{mult})$ (right wires).

Putting it Together. We are now able to refine the strawman protocol into a functional protocol. When evaluating a circuit layer, the players run a protocol to simulate differing-operation packed secret sharing, by evaluating each gate as both an addition gate and multiplication gate. Then, the leader runs the realignment procedure to prepare vectors that are appropriate for the next layer of computation. Finally, the leader secret shares these new vectors, distributing them to all players, and computing the next layer can commence. Conceptually, the protocol uses the leader to "unpack" and "repack" the shares to simultaneously satisfy both requirements of SIMD computation.

Leveraging Circuits with Highly Repetitive Structure. Until this point, we have been using the masking primitive imprecisely, assuming that it could accommodate the procedural changes discussed above without modification. This however, is not the case. Because we need to mask and unmask values while they are in a packed form, *the masks themselves must be generated and handled in packed form.*

Consider the example vectors used to describe differing-operation packed secret sharing, trying to compute $(x_1y_1, x_2 + y_2, x_3y_3)$ given (x_1, x_2, x_3) and (y_1, y_2, y_3). If the same mask (r_1, r_2, r_3) is used to mask both the sum and product of these vectors, privacy will not hold; for example, the leader will open the values $x_1 + y_1 + r_1$ and $x_1y_1 + r_1$, and thus learn something about x_1 and y_1. If (r_1, r_2, r_3) is used to mask addition and (r'_1, r'_2, r'_3) is used for multiplication, there is privacy, but it is unclear how to unmask the result. The shared vector distributed by the leader will correspond to $(x_1y_1 + r_1, x_2 + y_2 + r'_2, x_3y_3 + r_3)$ and the random values cannot be removed with only access to (r_1, r_2, r_3) and (r'_1, r'_2, r'_3). To run the realignment procedure,

the same problem arises: the unmasking vectors must have a different structure than the masking vectors, with their relationship determined by the structure of the next circuit layer.

We overcome this problem by making modifications to the batched randomness generation procedure. Instead of generating structurally identical masking and unmasking shares, we instead use the circuit structure to permute the random inputs used during randomness generation so we get outputs of the right form. In the example above, the players will collectively generate the *masking vectors* (r_1, r_2, r_3) and (r'_1, r'_2, r'_3), where each entry is sampled independently at random. The players then generate the *unmasking vector* (r_1, r'_2, r_3) by permuting their inputs to the generation algorithm. For a more complete description of this subprotocol, see Section C.

However, recall that it is critical for efficiency that we generate all randomness in *batches*. By permuting the inputs to the randomness generation algorithm, we get $\Theta(n)$ masks that are correctly structured *for a particular part of the circuit structure*. If this particular structure occurs only once in the circuit, only one of the $\Theta(n)$ shares can actually be used during circuit evaluation. In the worst case, if each circuit substructure is *unique*, the resulting randomness generation phase requires $O(n|\mathsf{C}|)$ communication complexity.

This is where the requirement for highly repetitive circuits becomes relevant. This class of circuits guarantees that (1) the circuit layers are wide enough that using packed secret sharing with vectors containing $\Theta(n)$ elements is appropriate, and (2) all $\Theta(n)$ shares of random values generated during the batched randomness generation phase can be used during circuit evaluation. We note that this is a rather simplified version of the definition, we give a formal definition of such circuits in Section 4.2.

Non-interactive packed secret sharing from traditional secret shares. Another limitation of the strawman protocol presented above is that the circuit must ensure that all inputs from a single party can be packed into a single packed secret sharing at the beginning of the protocol. We devise a novel strategy that allows parties to secret share each of their inputs individually using regular secret sharing. Parties can then *non-interactively* pack the appropriate inputs according to the circuit structure. This strategy can also be used to efficiently *switch* to protocols $O(n|\mathsf{C}|)$ protocols when parts of the circuit lack highly repetitive structure; the leader omits the repacking step, and the parties compute on traditional secret share until the circuits becomes highly repetitive, at which point they non-interactively re-packing any wire values (see Section 4.4).

Existing $O(|\mathsf{C}|)$ protocols like [DIK10] do not explicitly discuss how their protocol handles this input scenario. We posit that this is because there are generic transformations like embedding switching networks at the bottom of the circuit that allow any circuit to be transformed into a circuit in which a player's inputs can be packed together. Unsurprisingly, these transformations significantly increase the size of the circuit. Since [DIK10] is primarily concerned with asymptotic efficiency, such circuit modification strategies are sufficient for their work.

Comparison with [DIK10]. We briefly recall the strategy used in [DIK10], in order to overcome the limitations of working with packed secret sharing that we discussed earlier. They present a generic transformation that transforms any circuit into a circuit that satisfies the following properties:

1. The transformed circuit is layered and each layer only consists of one type of gates.
2. The transformed circuit is such that, when evaluating it over packed secret shares, there is never a need to permute values across different vectors/blocks that are secret shared. While the values within a vector might need to be permuted during circuit evaluation, the transformed circuit has a nice property that only $\log \ell$ (where ℓ is the size of the block) such permutations are needed throughout the circuit.

It is clear that the first property already gets around the first limitation of packed secret sharing. The second property partly resolves the *realignment* requirement from a packed secret sharing scheme by only requiring permutations within a given vector. This is handled in their protocol by generating permuted random blocks that are used for masking and unmasking in the multiplication sub-protocol. Since only $\log \ell$ different permutations are required throughout the protocol, they are able to get significant savings by generating random pairs corresponding to the same permutation in *batches*. Our "unpacking" and "repacking" approach can be viewed as a generalization of their technique, in the sense that we enable permutation and duplication of values across different vectors by evaluating the entire layer in one shot.

As noted earlier, this transformation introduces significant overhead to the size of the circuit, and is the primary reason for the large multiplicative and additive terms in the overall complexity of their protocol. As such, it is unclear how to directly use their protocol to compute circuits with highly repetitive structures, while skipping this circuit transformation step. This is primarily because these circuits might not satisfy the first property of the transformed circuit. Moreover, while it is true that the number of possible permutations required in such circuits are very few, they might require permuting values across different vectors, which cannot be handled in their protocol.

2.3 Malicious Security

Significant work has been done in recent years to build compilers that take semi-honest protocols that satisfy common structures and produce efficient malicious protocols, most notably in the "additive attack paradigm" described in [GIP+14]. These semi-honest protocols are secure *up to additive attacks*, that is any adversarial strategy is only limited to injecting additive errors onto each of the wires in the circuit that are independent of the "actual"wire values. The current generation of compilers for this class of semi-honest protocols, exemplified by [CGH+18,NV18,FL19,GSZ20], introduce only a small multiplicative overhead (e.g., 2 in the case of [CGH+18]) and require only a constant number of additional rounds to perform a single, consolidated check

Genkin et al. showed in [GIP15] (with additional technical details in [Gen16]) that protocols leveraging packed secret sharing schemes do not satisfy the structure required to leverage the compilers designed in the "additive attack paradigm." Instead, they show that most semi-honest protocols that use packed secret sharing are secure up to linear errors, that is the adversary can inject errors onto the output wires of multiplication gates that are *linear functions* of the values contained in the packed sharing of input wires to this gate. We observe that this also holds true for our semi-honest protocol. They present a malicious security compiler for such protocols that introduces a small multiplicative overhead.

To achieve malicious security, we add a new consolidated check onto our semi-honest protocol, reminiscent of the check for circuits over small-fields presented in Section 5 of [CGH+18]. The resulting maliciously secure protocol has approximately 2.3 times the complexity of our semi-honest protocol (depending on the choice of ϵ), plus a constant sized, consolidated check at the end – for the first time matching the efficiency of the compilers designed for protocols secure up to additive attacks.

As in [CGH+18], we run two parallel executions of the circuit, maintaining the invariant that for each packed set of wires $z = (z_1, z_2, \ldots, z_\ell)$ in C the parties also compute $z' = rz = (rz_1, rz_2, \ldots, rz_\ell)$ for a global, secret scalar value r. Once the players have shares of both z and z' for each wire in the circuit, we generate shares of random vectors $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_\ell)$ (one for each packed sharing vector in the protocol) using a malicious secure sub-protocol and reconstruct the value r. The parties then interactively verify that $r * \alpha * z = \alpha * z'$. Importantly, this check can be carried out simultaneously for all packed wires in the circuit, *i.e.*

$$r * \sum_{i \in C} \alpha_i * z_i = \sum_{i \in C} \alpha_i * z'_i$$

This simplified check relies heavily on the malicious security of the randomness generation sub-protocol. Because of the structure of linear attacks and the fact that α was honestly secret-shared, multiplying z and z' with α injects linear errors chosen by the adversary that are monomials in α only. That is, the equation becomes

$$r * \sum_{i \in C} (\alpha_i * z_i + E(\alpha)) = \sum_{i \in C} (\alpha_i * z'_i + E'(\alpha))$$

for adversarially chosen linear functions E and E'. Because α is independent of r and r is applied to the left hand side of this equation only at the end, this check will only pass if $r * E(\alpha) = E'(\alpha)$. For any functions $E(\cdot), E'(\cdot)$ this only happen if either (1) both are the zero function (in which case there are no errors), or (2) with probability $\frac{1}{|\mathbb{F}|}$. Hence, this technique can also be used with packed secret sharing to get an efficient malicious security compiler.

3 Preliminaries

Model and Notation. We consider a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ in which each party provides inputs to the functionality, participates in the evaluation protocol, and receives an output. We denote an arbitrarily chosen special party P_{leader} for each layer (of the circuit) who will have a special role in the protocol; we note that the choice of P_{leader} may change in each layer to better distribute computation and communication. Each pair of parties are able to communicate over point-to-point private channels.

We consider a functionality that is represented as an arithmetic circuit C over a field $\mathbb{F}$, with maximum width w and total total depth d. We visualize the circuits in a bottom-up setting (like in Merkle trees), where the input gates are at the bottom of the circuit and the output gates are at the top. As we will see later in the definition of highly repetitive circuits, we work with *layered circuits*, which comprise of layers such that the output of layer i are only used as input for the gates in layer $i+1$.

We consider security against a static adversary Adv that corrupts $t \leq n(\frac{1}{2} - \frac{2}{\epsilon})$ players, where ϵ is a tunable parameter of the system. As we will be working with both a packed secret sharing scheme (see Section A.3) and a slightly modified version of regular threshold secret sharing scheme (see Section A.2), we require additional notation. We denote the packing constant for our protocol as $\ell = \frac{n}{\epsilon}$. Additionally, we will denote the threshold of our packed secret sharing scheme as $D = t + 2\ell - 1$. We will denote vectors of packed values with **bold** alphabets, for instance $\mathbf{x}$. Packed secret shares of a vector $\mathbf{x}$ with respect to degree D are denoted $[\mathbf{x}]$ and with respect to degree $n-1$ as $\langle \mathbf{x} \rangle$. We let $e_1, \ldots, e_\ell$ be the fixed x-coordinates on the polynomial used for packed secret sharing, where the ℓ secrets will be stored, and $\alpha_1, \ldots \alpha_n$ be the fixed x-coordinates corresponding to the shares of the parties. For regular threshold secret sharing, we will only require shares w.r.t. degree $t+\ell$. We use the *square bracket* notation to denote a secret sharing w.r.t. degree $t+\ell$. We note that we work with a slightly modified sharing algorithm of the Shamir's secret sharing scheme (see Section A.2 for details). We denote the Vandermonde matrix $\mathbf{V}_{n,(n-t)} \in \mathbb{F}^{n \times (n-t)}$. which is defined as follows:

$$\begin{bmatrix} 1 & \gamma_1 & \ldots & \gamma_1^{n-t-1} \\ 1 & \gamma_2 & \ldots & \gamma_2^{n-t-1} \\ \cdot & \cdot & \ldots & \cdot \\ \cdot & \cdot & \ldots & \cdot \\ 1 & \gamma_n & \ldots & \gamma_n^{n-t-1} \end{bmatrix}$$

where $\gamma_1, \ldots, \gamma_n \in \mathbb{F}$ are n distinct non-zero elements. In some cases, we also use a hyper-invertible matrix as defined in [BTH08] and denote it by $\mathbf{H}_{n,n} \in \mathbb{F}^{n \times n}$.

Paper Organization In Section 4 we define the class of highly repetitive circuits and give some natural examples of such circuits. Section 5.3, we describe our non-interactive protocol for packing regular shares. Section 8 gives a construction of our semi-honest and maliciously secure protocols. In Section 9, we give details of our implementation and present an extensive comparison with prior work.

4 Highly Repetitive Circuits

In this section, we formalize the class of highly repetitive circuits and discuss some examples of naturally occurring highly repetitive circuits.

4.1 Wire Configuration

We start by formally defining a *gate block*, which is the minimum unit over which we will reason.

Definition 1 (Gate Block). *We call a set of j gates that are all on the same layer, a gate block. We say the size of a gate block is j.*

An additional non-standard functionality we require is an explicit wire mapping function. Recall from the technical overview that the leader must repack values according to the structure of the next layer. To reason formally over this procedure, we define the function $\mathsf{WireConfiguration}$, which takes in two blocks of gates block_{m+1} and block_m, such that the output wires of the gates in block_m feed as input to the gates in block_{m+1}. $\mathsf{WireConfiguration}$ outputs two ordered arrays $\mathsf{LeftInputs}$ and $\mathsf{RightInputs}$ that contain the indices corresponding to the left input and right input of each gate in block_{m+1} respectively. In general, we can say that $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$ will output a correct alignment for block_{m+1}. This is because for all values $j \in [|\mathsf{block}_{m+1}|]$, if the values corresponding to the wire $\mathsf{LeftInputs}[j]$ and $\mathsf{RightInputs}[j]$ are aligned, then computing block_{m+1} is possible. We describe the functionality for $\mathsf{WireConfiguration}$ in Figure 2. It is easy to see that the blocks $\mathsf{block}_{m+1}, \mathsf{block}_m$ must lie on consecutive layers in the circuit. We say that a pair of gate blocks is *equivalent* to another pair of gate blocks, if the outcome of $\mathsf{WireConfiguration}$ on both pairs is identical.

The Function $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$

1. Initialize two ordered arrays $\mathsf{LeftInputs} = [\ ]$ and $\mathsf{RightInputs} = [\ ]$, each with capacity $|\mathsf{block}_{m+1}|$.
2. For a gate g, let $l(g) = (j, \mathsf{type})$ denote the index j and type of the gate in block block_m that feeds the left input of g. Similarly, let $r(g) = (j, \mathsf{type})$ denote the right input gate index and type of g. For gates with fan-in one, *i.e.* relay gates, $r(g) = 0$. For each gate g_j in block_{m+1}, we set
   — $\mathsf{LeftInputs}[j] = l(g_j)$
   — $\mathsf{RightInputs}[j] = r(g_j)$
3. Output $\mathsf{LeftInputs}, \mathsf{RightInputs}$.

Fig. 2: The function $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$ that computes a proper alignment for computing block_{m+1}

4.2 (A, B)-Repetitive Circuits

With notation firmly in hand, we can now formalize the class of (A, B)-repetitive circuits, where A, B are the parameters that we explain next. Highly repetitive circuits are a subset of (A, B)-repetitive circuits, which we will define later.

We define an (A, B)-repetitive circuit using a partition function part that decomposes the circuit into blocks of gates, where a block consists of gates on the same layer. Let $\{\mathsf{block}_{m,j}\}$ be the output of this partition function, where m indicates the layer of the circuit corresponding to the block and j is its index within layer m. Informally speaking, an (A, B)-repetitive circuit is one that satisfies the following properties:

1. Each block $\mathsf{block}_{m,j}$ consist of at least A gates.
2. For each pair $(\mathsf{block}_{m,j}, \mathsf{block}_{m+1,j})$, all the gates in $\mathsf{block}_{m+1,j}$ only take in wires that are output wires of gates in $\mathsf{block}_{m,j}$. And the output wires of all the gates in $\mathsf{block}_{m,j}$ only go an input to the gates in $\mathsf{block}_{m+1,j}$.
3. For each pair $(\mathsf{block}_{m,j}, \mathsf{block}_{m+1,j})$, there exist at least B other pairs with identical wiring between the two blocks.

We now give a formal definition.

Definition 2 ((A, B)-Repetitive Circuits). *We say that a layered circuit* C *with depth* d *is called an* (A, B)*-repetitive circuit if there exists a value* $\sigma \geq 1$ *and a partition function* part *which on input* layer_m *(*m^{th} *layer in* C*), outputs disjoint blocks of the form*

$$\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m}),$$

such that the following holds, for each $m \in [d], j \in [\sigma]$:

1. ***Minimum Width:*** *Each $\mathsf{block}_{m,j}$ consists of at least A gates.*
2. ***Bijective Mapping:*** *All the gates in $\mathsf{block}_{m,j}$ only take inputs from the gates in $\mathsf{block}_{m-1,j}$ and only give outputs to gates in $\mathsf{block}_{m+1,j}$.*
3. ***Minimum Repetition:*** *For each $(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$, there exist pairs $(m_1, j_1) \neq (m_2, j_2) \neq \ldots \neq (m_B, j_B) \neq (m, j)$ such that for each $i \in [B]$, $\mathsf{WireConfiguration}(\mathsf{block}_{m_i+1,j_i}, \mathsf{block}_{m_i,j_i}) = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.*

Intuitively, this says that a circuit is built from an arbitrary number of gate blocks with sufficient size, and that all blocks are repeated often throughout the circuit. Unlike the layer focused example in the introduction, this definition allows layers to comprise of multiple blocks. In fact, these blocks can even *interact* by sharing input values. The limitation of this interaction, captured by the $\mathsf{WireConfiguration}$ check, is that the interacting inputs must come from predictable indices in the previous layer and must have the same gate type.

We also consider a relaxed variant of (A, B)-repetitive circuits, which we call (A, B, C, D)-repetitive circuits. These circuits differ from (A, B)-repetitive circuits in that they allow for a relaxation of the minimum width and repetition requirement. In particular, in an (A, B, C, D)-repetitive circuit, it suffices for all but C blocks to satisfy the minimum width requirement and similarly, all but D blocks are required to satisfy the minimum repetition requirement. In this work, we focus on the following kind of (A, B, C, D)-repetitive circuits.

Definition 3 (Highly Repetitive Circuits). *We say that (A, B, C, D)-repetitive circuits are highly repetitive w.r.t. n parties, if $A, B \in \Omega(n)$ and C, D are some constants.*

We note that defining a class of circuits w.r.t. to the number of parties that will evaluate the circuit might a priori seem unusual. However, this is common throughout the literature attempting to achieve $O(|\mathsf{C}|)$ MPC that use packed secret sharing. For example, the protocols in [DIK+08,DIK10,GIP15] achieve $\tilde{O}(|C|)$ communication for circuits that are $\Omega(n)$ gates wide. Similarly, our work achieves $O(|C|)$ communication and computation for circuits that are $(\Omega(n), \Omega(n), C, D)$-repetitive, where C and D are constants. Alternatively, if the number of input wires are equal to the number of participating parties, we can re-phrase the above definition w.r.t. the number of input wires in a circuit.

It might be useful to see the above definition as putting a limit on the number of parties for which a circuit is highly repetitive: any (A, B, C, D)-repetitive circuit, is highly repetitive for upto $\mathtt{min}(O(A), O(B))$ parties. While our MPC protocol can work for any (A, B, C, D)-repetitive circuit, it has $O(|\mathsf{C}|)$ complexity only for highly repetitive circuits. In the next subsection we give examples of such circuits that are highly repetitive for a reasonable range of parties.

For the remainder of this paper, we will use w denote the maximum width of the circuit C, w_m to denote the width of the mth layer and $w_{m,j}$ to denote the width of $\mathsf{block}_{m,j}$.

4.3 Examples of Highly Repetitive Circuits

We highlight 3 functionalities with circuit representations that are part of the highly repetitive circuit class. First, we describe machine learning circuits, focusing on training algorithms that leverage gradient decent. Then, we discuss cryptographic hash functions like SHA256 and block ciphers like AES.

Machine Learning. Machine learning algorithms extract trends from large datasets to facilitate accurate prediction in new, unknown circumstances. Training can been viewed as an optimization problem, in which the model attempts to find internal parameters that minimizes the error between its predictions and ground truth. A common family of algorithms for minimizing this error is called "gradient decent." Starting with random internal parameters, the algorithm iteratively reduces the error by making a small, greedy changes. When run without privacy, the algorithm terminates when it converges (*i.e.* the marginal decrease in error

Table 1: Size of the highly repetitive circuits we consider in this work. We compile these functions into $\mathbb{F}_2$ circuits using Frigate [MGC+16] (containerized by [HHNZ19]). The 64 iterations of the compression function for SHA256 comprise 77% of the gates and the round function of AES comprises 88% of the gates. Both of these metrics are computed for a single block on input.

Circuit	Gates ($\mathbb{F}_2$)	Iterative Loops	Gates per Loop	Percent Repeated Structure
SHA256 (1 Block)	119591	64	1437	77%
AES128 (1 Block)	7458	10	656	88%
Gradient Descent	—	≥ 10000	—	$\sim 100\%$

is zero). However, because MPC computation must be data oblivious, the number of iterations must be selected *before execution* and must cover the worst case scenario. Different versions of this algorithm are used to train simple models, like linear regression, or more complex and powerful models, like neural networks. For a more complete description of gradient decent training algorithms, and their adaptation to MPC, see [MR18].

The exact number of gates in the circuit representation of privacy-preserving model training is difficult to calculate from prior work. In one of the few concrete estimates, Gascón et al. [GSB+16] realize coordinate gradient decent training algorithms with approximately 10^{11} gates. As noted in [MZ17], the storage requirement for this circuit would be 3000GB. Subsequent work stopped estimating gate counts altogether, instead building a library of sub-circuits that can be loaded as needed. As the amount of data used to train models continues to grow, circuits sizes will continue to increase. While we are not able to accurately estimate the number of gates for this kind of circuit, we can still establish that their structure is highly repetitive. For instance, the gradient decent algorithm consists of nothing but iterations of the same functionality. In the implementation of Mohassel et al. [MR18], the default configuration for training is 10000 iterations, clearly enough repeated depth to accommodate massive number of players. Indeed, in the worst case the depth of a gradient decent algorithm must be linear in the input size. This is because gradient decent usually uses a *batching* technique, in which only a subset of the data is used for any given iteration. However, as all the algorithm wants is to accommodate as much new data as possible, the number of batches should be linear in the input size.

The width of gradient decent training algorithms is usually roughly proportional to the dimension of the dataset. For most interesting applications of machine learning, high dimensional data is normal. If a particular application does not have high enough dimension to allow massive number of parties to participate in the protocol, we note that parallelism can be leveraged. Specifically, gradient decent training algorithms usually use a *random restart* strategy to avoid getting trapped at local minima. These independent runs of the algorithm can be run in parallel, making the circuit quite wide. Some final logic may be added at the end to select the output from the iterations that produced optimal internal parameters.

Cryptographic Hash Functions. All currently deployed cryptographic hash functions rely on iterating over a round function. This round function typically has a diffusion property such that, after many invocations, it is widely considered impossible to invert. Importantly for our purposes, each iteration of the round function is (typically) *structurally identical*. Moreover, the vast majority of the gates in the circuit representation of a hash function are contained within the iterations of the round function. As a concrete study of such a cryptographic hash function, we consider SHA256 [NIS02]. SHA256 is one of the most widely deployed hash functions; given its common use in applications like Bitcoin [Nak08] and ECDSA [GFD09], SHA256 is an important building block of MPC applications. SHA256 contains 64 rounds of its inner function, with other versions that use larger block size containing 80 rounds.

To measure the proportion of the SHA256 circuit that is contained within the iterated round function, we implement a Frigate [MGC+16] compatible SHA256 description for hashing a single block of input. While our protocol is intended for arithmetic circuits, but there are no well tested arithmetic circuit compilers and

our protocol can be adapted to binary field. As can be seen in Table 1, 77% of all the gates in the compiled SHA256 are repeated structure, that structure repeating at least 64 times.

We note that these results were for hashing only a single block of input. When hashing a single block of input, there are gates to handle initialization and output, comprising the remaining 23% of gates. However, it is unlikely that an MPC with hundreds or thousands of players will compute only a single block of SHA256; it is more plausible that each participating player will contribute additional data, for $O(n)$ total blocks. These additional blocks of input do not contain the overhead, so all the additional gates will comprise repeated structure. For instance, if there are as few as 10 blocks of input, the circuit is already 97% repeated structure.

If we consider the case where the number of blocks of input is proportional to the number of player, all that remains to argue is that the width of the circuit is sufficient that each gate block is sufficiently large. As mentioned, there are no good arithmetic compilers available, so it is difficult to argue about the width of the arithmetic circuit computing the functionality SHA256. We note that the width of a block is 512 bits. If width is proportional to this, it is very plausible to say hundreds of players could compute this functionality. However, when computing over a larger field, there may not be enough gates in each layer. As such, we note that there are many common applications which require many *parallel* iterations of hash functions. For instance, if players wish to compute a Merkle tree over their inputs, the resulting circuit will naturally satisfy our requirements.

Block Ciphers. Modern block ciphers, similar to cryptographic functions, are iterative by nature. Advanced Encryption Standard, the block cipher on which we focus, uses either 10, 12, or 14 iterations of its round function, depending on the key length used. The round function is comprised of a substitution step, a shifting step, a mixing step, with all but one iteration containing all of these steps. Again, this repeated structure allows the pre-processing phase of our protocol to be run very efficiently. Performing a similar analysis as with SHA256, we identified that 88% of the gates in AES128 are part of this repeated structure when encrypting a single block of input. Just as with hash functions, more blocks of input lead to increased percentage repeated structure. With 10 blocks of input, 98% of the gates are repeated structure.

As with hash functions, we note that width may be a concern for applying our protocol. However, computing many parallel encryptions is also a common task. For instance, if players wish to encrypt or decrypt a disk image, encrypting under multiple keys is common. These different sectors can be evaluated in parallel, giving sufficient structure.

4.4 Protocol Switching for Circuits with Partially Repeated Structure

Hash functions and symmetric key cryptography are not comprised of 100% repeated structure. When structure is not repeated, the batched randomness generation step cannot be run efficiently. In the worst case, if a particular piece of structure is only present once in the circuit, $O(n^2)$ messages will be used to generate only a single packet secret share of size $O(n)$. If $0 \leq p \leq 1$ is the fraction of the circuit that is repeated, our protocol has efficiency $O(p|\mathsf{C}| + (1-p)n|\mathsf{C}|)$.

We note that our protocol has worse constants than [CGH+18] and [FL19] when run on the non-repeated portion of the circuit. Specifically, our protocol requires communication for all gates, rather than just multiplication gates. As we are trying to push the constants as low as possible, it would be ideal to run the most efficient known protocols for the portions of the circuit that are linear in the number of players. To do this, we note that our protocol can support mid-evaluation protocol switching.

Recall our simple non-interactive technique to transform normal secret shares into packed secret shares, presented in Section 5.3. This technique can be used in the middle of protocol execution to switch between a traditional, efficient, $O(n|\mathsf{C}|)$ protocol and our protocol. Once the portion of the circuit without repeated structure is computed using another efficient protocol, the players can pause to properly structure their secret shares and non-interactively pack them. The players can then evaluate the circuit using our protocol. If another patch of non-repeated structure is encountered, the players can use the leader to reconstruct and re-share normal shares as necessary. Importantly, because all of these protocols are linear, it is still possible to use the malicious security compiler of [CGH+18].

5 Input Sharing Phase

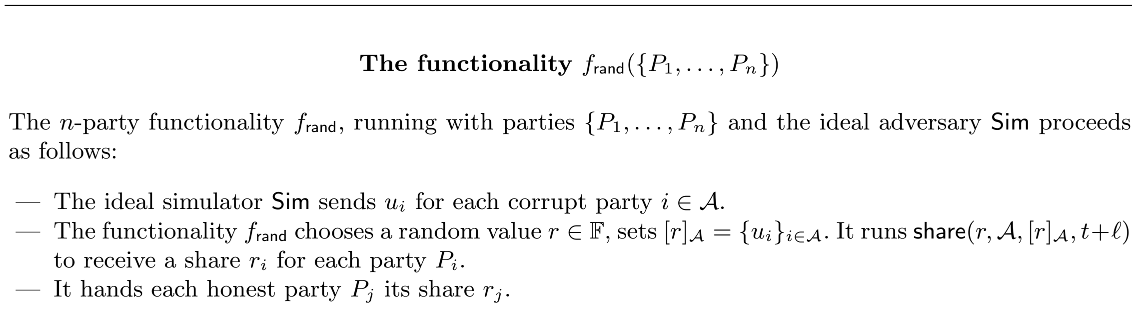
In this section, we present the sub-protocols/functionalities that will be used for secret sharing inputs in our main protocols. We begin by describing the functionality for generating (regular) shares for random values in Section 5.1. Then in Section 5.2, we show how the parties can use the previous functionality for computing (regular) shares of their inputs. Then in Section 5.3, we describe a non-interactive transformation that allows a set of parties holding shares corresponding to ℓ secrets, to compute a single packed secret sharing of the vector containing those ℓ secrets. Finally, in Section 5.4, we show how the above protocols can be combined to enable parties to obtain packed secret sharings of their inputs.

5.1 Generating Shares of Random Values

In this section, we describe a protocol π_{rand} for generating (regular) shares of a batch of random and independently chosen values (this is identical to the protocol proposed in [DN07]). In our main protocol, π_{rand} will help us robustly share inputs.

This protocol either outputs honestly computed (regular) shares of random values or it outputs $\perp$. It makes use of the regular Shamir's secret sharing scheme along with an $n \times n$ hyper-invertible matrix. First, each party samples a random value and (regular) secret shares it among the other parties. The parties compute n linear combinations of these shares using the Vandermonde matrix. The parties then open t sets of resulting shares to all the parties, who locally verify the correctness of these shares. If all n parties are happy with their checks, the remaining $n-t$ shares are output by the protocol. If the check succeeds, then the hyper-invertability property of guarantees that the remaining $n-t$ shares are random and honestly generated. We now proceed to formally define the f_{rand} functionality and then describe a protocol that securely computes $n-t$ instantiations of f_{rand} with abort. As discussed earlier, here we will work with a slightly modified sharing algorithm of the Shamir's secret sharing scheme (see Section A.2 for details).

The ideal functionality realized by this protocol is described in Figure 3. Since the adversary can choose its own shares in the protocol, similar to Chida et. all [CGH+18], we let adversary send shares of the corrupted parties to the ideal functionality. A formal description of the protocol π_{rand} that securely realizes this functionality appears in Section B.1.



The functionality $f_{\mathsf{rand}}(\{P_1, \ldots, P_n\})$

The n-party functionality f_{rand}, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

— The ideal simulator Sim sends u_i for each corrupt party $i \in \mathcal{A}$.
— The functionality f_{rand} chooses a random value $r \in \mathbb{F}$, sets $[r]_{\mathcal{A}} = \{u_i\}_{i \in \mathcal{A}}$. It runs $\mathsf{share}(r, \mathcal{A}, [r]_{\mathcal{A}}, t+\ell)$ to receive a share r_i for each party P_i.
— It hands each honest party P_j its share r_j.

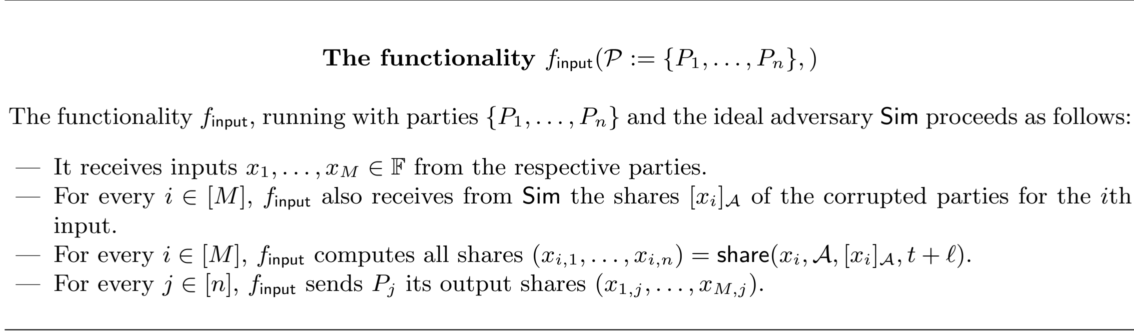
Fig. 3: Random share generation functionality

5.2 Secret Sharing of Inputs

In this section, we describe a well known protocol π_{input} for generating honest shares of each parties' inputs. We borrow much of the language from Chida et. al in [CGH+18] for this description. This sub-protocol will be used in our protocol to give robust sharings of inputs. Note that because we operate on packed secret

shares, this protocol alone is not sufficient to prepare inputs for evaluation. We describe a non-interactive way of transforming these robust shares into (robust) packed secret shares in the next section.

For each input x_i belonging to party P_i, the parties invoke f_{rand} to generate a random sharing $[r_i]$. They open the value of r to the designated owner P_i of x_i. P_i reconstructs r_i, computes $x_i - r_i$ and sends $x_i - r_i$ to all the parties. Each party then adds this value to its respective share of r_i. Since f_{rand} ensures that $[r]$ is an honest sharing of r, this in turn ensures the sharing of x_i is also honest. The ideal functionality realized by this protocol is described in Figure 4. A formal description of the protocol appears in Section B.2.



The functionality $f_{\mathsf{input}}(\mathcal{P} := \{P_1, \ldots, P_n\},)$

The functionality f_{input}, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

— It receives inputs $x_1, \ldots, x_M \in \mathbb{F}$ from the respective parties.
— For every $i \in [M]$, f_{input} also receives from Sim the shares $[x_i]_{\mathcal{A}}$ of the corrupted parties for the ith input.
— For every $i \in [M]$, f_{input} computes all shares $(x_{i,1}, \ldots, x_{i,n}) = \mathsf{share}(x_i, \mathcal{A}, [x_i]_{\mathcal{A}}, t + \ell)$.
— For every $j \in [n]$, f_{input} sends P_j its output shares $(x_{1,j}, \ldots, x_{M,j})$.

Fig. 4: Secret sharing of inputs functionality

5.3 A Non-Interactive Protocol for Packing Regular Secret Shares

We now describe a novel, non-interactive transformation that allows a set of parties holding shares corresponding to ℓ secrets $[s_1], \ldots, [s_\ell]$ to compute a single packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$. This protocol makes a non-black-box use of Shamir secret sharing to accomplish this packing without interaction. As discussed in the technical overview, to achieve efficiency, our protocol computes over packed shares. But, if each player follows the naïve strategy of just packing all their own inputs into a single vector, the values may not be properly aligned for computation. This non-interactive functionality lets players simply share their inputs using f_{input}, which is a simple input sharing functionality based on Shamir secret sharing (see Section B.2), and then locally pack the values in a way that guarantees alignment.

Let $p_1, \ldots, p_\ell$ be the degree $t+\ell$ polynomials that were used for secret sharing secrets $s_1 \ldots, s_\ell$ respectively such that each $p_i(z)$ (for $i \in [\ell]$) is of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial. Then each party P_j (for $j \in [n]$) holds shares $p_1(\alpha_j), \ldots, p_\ell(\alpha_j)$.

Given these shares, each party P_j computes a packed secret share of the vector $(s_1, \ldots, s_\ell)$ as follows:

$$\mathcal{F}_{\mathsf{SS-to-PSS}}(\{p_i(\alpha_j)\}_{i \in [\ell]}) = \sum_{i=1}^{\ell} p_i(\alpha_j) L_i(\alpha_j) = p(\alpha_j)$$

where $L_i(\alpha_j) = \prod_{j=1, j \neq i}^{\ell} \frac{(\alpha_i - e_j)}{(e_i - e_j)}$ is the Lagrange interpolation constant and p corresponds to a new degree $D = t + 2\ell - 1$ polynomial for the packed secret sharing of vector $\mathbf{v} = (s_1, \ldots, s_\ell)$.

Lemma 1. *For each $i \in [\ell]$, let $s_a \in \mathbb{F}$ be secret shared using a degree $t + \ell$ polynomial p_i of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial and $e_1, \ldots, e_\ell$ are some pre-determined field elements. Then for each $j \in [n]$, $\mathcal{F}_{\mathsf{SS-to-PSS}}(\{p_i(\alpha_j)\}_{i \in [\ell]})$ outputs the jth share corresponding to a valid packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$, w.r.t. a degree-$D = t + 2\ell - 1$ polynomial.*

Proof. For each $i \in [\ell]$, let $p_i(z)$ be the polynomial used for secret sharing the secret s_i. We know that $p_i(z)$ is of the form

$$p_i(z) = s_i + q_i(z) \prod_{j=1}^{\ell} (z - e_j),$$

where q_i is a degree t polynomial. Let $p'_i(z) = q_i(z) \prod_{j=1}^{\ell}(z - e_j)$ and let $p(z)$ be the new polynomial corresponding to the packed secret sharing. From the description of $\mathcal{F}_{\mathsf{SS-to-PSS}}$, it follows that:

$$\begin{aligned}
p(z) =& \sum_{i=1}^{\ell} p_i(z) L_i(z) = \sum_{i=1}^{\ell} p'_i(z) L_i(z) + s_i L_i(z) \\
=& \sum_{i=1}^{\ell} p'_i(z) \prod_{j=1, j\neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)} + \sum_{i=1}^{\ell} s_i L_i(z) \\
=& \sum_{i=1}^{\ell} q_i(z) \prod_{j=1, j\neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)} \prod_{j=1}^{\ell} (z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z)
\end{aligned}$$

$$\text{Let } q'_i(z) = q_i(z) \prod_{j=1, j\neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)}$$

$$\begin{aligned}
p(z) =& (q'_1(z) + \ldots + q'_\ell(z)) \prod_{j=1}^{\ell} (z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z) \\
=& q(z) \prod_{j=1}^{\ell} (z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z)
\end{aligned}$$

where $q(z) = q'_1(z) + \ldots + q'_\ell(z)$ is a degree $t + \ell - 1$ polynomial and hence $p(z)$ is a degree $D = t + 2\ell - 1$ polynomial. It is now easy to see that for each $i \in [\ell]$, $p(e_i) = s_i$. Hence $\mathcal{F}_{\mathsf{SS-to-PSS}}$ computes a valid packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$.

5.4 Packed Secret Sharing of Inputs

We now arrive at a subprotocol that will be invoked directly in our protocol execution. This functionality takes in the individual inputs of the players and outputs a packed secret sharing of these inputs. Using the circuit information, players can run $\mathsf{WireConfiguration}(\mathsf{block}_{0,j}, \mathsf{block}_{1,j})$ for each $j \in [\sigma]$ to determine the alignment of vectors required to compute the first layer of the circuit. Because each $\mathsf{block}_{1,j}$ in the circuit contains $w_{1,j}/\ell$ gates, the protocol outputs $2w_1/\ell = \sum_{j\in[\sigma]} w_{1,j}$ properly aligned packed secret shares, each containing ℓ values. This functionality makes use of our non-interactive packing protocol described in Section 5.3.

A formal description of the ideal functionality for this subprotocol appears in Figure 5. A detailed description of the subprotocol appears in Section B.3.

6 Circuit Evaluation Phase

In this section, we present the sub-protocols that will be used in the online evaluation of the circuit. In Section 6.1, we present our randomness generation sub-protocol that outputs packed shares of correlated random values, where the correlation is dictated by the configuration of the circuit. Then in Section 6.2, we present our main circuit evaluation subprotocol, that takes the random shares generated by the previous protocol and packed shares of input vectors output by the subprotocol from Section 5.4 to evaluate the circuit layer-wise.

The functionality $f_{\mathsf{pack-input}}(\mathcal{P} := \{P_1, \ldots, P_n\},)$

The functionality $f_{\mathsf{pack-input}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

— It receives inputs $x_1, \ldots, x_M \in \mathbb{F}$ from the respective parties and the layers $\mathsf{layer}_0, \mathsf{layer}_1$ from all parties.
— It computes $\{\mathsf{block}_{0,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(0, \mathsf{layer}_0)$ and $\{\mathsf{block}_{1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(1, \mathsf{layer}_1)$.
— For each $j \in [\sigma]$, it computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{1,j}, \mathsf{block}_{0,j})$.
— For each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$,

- Set $\mathbf{x}^{j,q} = (x_{\mathsf{LeftInputs}_j[i]})_{i\in\{(q-1)\ell+1,\ldots,q\ell\}}$ and $\mathbf{y}^{j,q} = (x_{\mathsf{RightInputs}_j[i]})_{i\in\{(q-1)\ell+1,\ldots,q\ell\}}$.
- Receives from Sim, the shares $[\mathbf{x}^{j,q}]_{\mathcal{A}}, [\mathbf{y}^{j,q}]_{\mathcal{A}}$ of the corrupted parties for the input vectors $\mathbf{x}_{j,q}, \mathbf{y}_{j,q}$.
- It computes shares $\mathbf{x}^{j,q} \leftarrow \mathsf{pshare}(\mathbf{x}^{j,q}, \mathcal{A}, [\mathbf{x}^{j,q}]_{\mathcal{A}}, D)$ and $\mathbf{y}^{j,q} \leftarrow \mathsf{pshare}(\mathbf{y}^{j,q}, \mathcal{A}, [\mathbf{y}^{j,q}]_{\mathcal{A}}, D)$ and sends them to the parties.

Fig. 5: Packed Secret sharing of all inputs functionality

6.1 Generating Correlated Random Packed Sharings

We now turn to the randomness generation protocol for our main construction. Recall from the technical overview that the packed secret sharings of random values must be generated according to the circuit structure. More specifically, the unmasking values (degree D shares) for some $\mathsf{block}_{m+1,j}$ must be aligned according to the output of $\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.

Before describing the protocol, we quickly note the number of shares that it generates, as it is somewhat non-standard. Let $w_{m,j}$ be the number of gates in $\mathsf{block}_{m,j}$ and $w_{m+1,j}$ be the number of gates in $\mathsf{block}_{m+1,j}$. As noted in the technical overview, our protocol treats each gate as though it performs *all* operations (relay, addition and multiplication). This lets the players evaluate different operations on each value over packed secret shares. Each of these operations must be masked with different randomness to ensure privacy. As such, the protocol generates $3w_{m,j}/\ell$ shares of uniform random vectors. To facilitate unmasking after the leader has run the realignment procedure, the protocol must generate shares of vectors with values selected from these $3w_{m,j}/\ell$ random vectors. This selection is governed by $\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$. Since there are $w_{m+1,j}$ gates in $\mathsf{block}_{m+1,j}$, the functionality will output $2w_{m+1,j}/\ell$ of these unmasking shares (with degree D). In total, these are $(3w_{m,j} + 2w_{m+1,j})/\ell$ packed secret sharings.

To summarize, this protocol has the following main steps:

1. The parties generate $3w_{m,j}/\ell$ uniform random vectors, corresponding to the values that will be used to mask the outputs of $\mathsf{block}_{m,j}$.
2. Parties compute $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$ to determine the required alignment of the correlated random vectors.
3. Parties use $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j$ and the gate information of $\mathsf{block}_{m,j}$ to select the appropriate values from step 1 for unmasking shares. This results in $2w_{m+1,j}/\ell$ vectors
4. Parties share these $(3w_{m,j} + 2w_{m+1,j})/\ell$ vectors using packed secret sharing and deal the resulting shares to all parties.
5. Parties use the Vandermonde matrix $\mathbf{V}_{n,(n-t)}$ to compute linear combinations of the shares they have received, and output the result.

We give a formal description of this subprotocol $\pi_{\mathsf{corr-rand}}$ in Section C.

The protocol $\pi_{\mathsf{eval}}(\{P_1, \ldots, P_n\})$

Input: The parties $\{P_i\}_{i\in[n]}$ hold packed secret sharings $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}], [\mathbf{x}_2^{j,q}], [\mathbf{y}_2^{j,q}]\right\}_{j\in[\sigma], q\in[w_{1,j}/\ell]}$ and for each $m \in [d]$, they hold configuration of layers layer_m and layer_{m+1}. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let $\mathsf{Unique} \subseteq \{(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})\}_{d\in[m], j\in[\sigma]}$ be such that for every pair $(\mathsf{block}_{a+1}, \mathsf{block}_a), (\mathsf{block}_{b+1,b}, \mathsf{block}_b) \in \mathsf{Unique}$, it holds that $\mathsf{WireConfiguration}(\mathsf{block}_{a+1}, \mathsf{block}_a) \neq \mathsf{WireConfiguration}(\mathsf{block}_{b+1}, \mathsf{block}_b)$.
Protocol: The parties proceed as follows:

— *Generating Correlated Randomness:* For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$, the parties run $\pi_{\mathsf{corr-rand}}$ to obtain packed secret shares $\{\{[\mathbf{r}_i^{q,\mathsf{left}}], [\mathbf{r}_i^{q,\mathsf{right}}]\}_{q\in[w_{a+1}/\ell]}, \{\langle\mathbf{r}_i^{q,\mathsf{mult}}\rangle, \langle\mathbf{r}_i^{q,\mathsf{add}}\rangle, \langle\mathbf{r}_i^{q,\mathsf{relay}}\rangle\}_{q\in[w_a/\ell]}\}_{i\in[n-t]}$, where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively. The parties then assign these shares to different blocks in the circuit based on the configuration of each block. In other words, we assume that at the end of this step for each $m \in [d], j \in [\sigma]$, the parties have the following shares:

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle\mathbf{r}_m^{j,q,\mathsf{mult}}\rangle, \langle\mathbf{r}_m^{j,q,\mathsf{add}}\rangle, \langle\mathbf{r}_m^{j,q,\mathsf{relay}}\rangle\}_{q\in[w_{m,j}/\ell]}$$

— *Layer-wise Evaluation:* Circuit evaluation proceeds layer-wise, where for each $m \in [d], j \in [\sigma]$, the parties proceed as follows:
- For each $q \in [w_{m,j}/\ell]$, the parties locally compute the following:
  $\langle\mathbf{x}_1^{j,q} \cdot \mathbf{y}_2^{j,q} + \mathbf{r}_m^{j,q,\mathsf{mult}}\rangle = [\mathbf{x}_1^{j,q}] \cdot [\mathbf{y}_2^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{mult}}\rangle$
  $\langle\mathbf{x}_1^{j,q} + \mathbf{y}_1^{j,q} + \mathbf{r}_m^{j,q,\mathsf{add}}\rangle = [\mathbf{x}_1^{j,q}] + [\mathbf{y}_1^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{add}}\rangle$
  $\langle\mathbf{x}_1^{j,q} + \mathbf{r}_m^{j,q,\mathsf{relay}}\rangle = [\mathbf{x}_1^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{relay}}\rangle$
- All the parties send their shares to the designated party P_{leader} for that layer.
- Party P_{leader} proceeds as follows:
  1. It reconstructs all the shares to get individual values $\{z_i^{j,\mathsf{mult}}, z_i^{j,\mathsf{add}}, z_i^{j,\mathsf{relay}}\}_{j\in[\sigma], i\in[w_{m,j}]}$. It then computes the values $z_i^{j,1}, \ldots, z_i^{j,w_m}$ on the outgoing wires from the gates in layer m as follows: For each $j \in [\sigma], i \in [w_{m,j}]$:
     * If gate $g_m^{j,i}$ is a multiplication gate, it sets $z^{j,i} = z_i^{j,\mathsf{mult}}$.
     * If gate $g_m^{j,i}$ is an addition gate, it sets $z^{j,i} = z_i^{j,\mathsf{add}}$.
     * If gate $g_m^{j,i}$ is a relay gate, it sets $z^{j,i} = z_i^{j,\mathsf{relay}}$.
  2. It then computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.
  3. For each $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$ each $i \in [\ell]$, let $e_{\mathsf{left}} = \mathsf{LeftInputs}[\ell \cdot (j-1) + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[\ell \cdot (j-1) + i]$, it sets $\overline{\mathbf{z}}^{j,q,\mathsf{left}}[i] = z^{j,e_{\mathsf{left}}}$ and $\overline{\mathbf{z}}^{j,q,\mathsf{right}}[i] = z^{j,e_{\mathsf{right}}}$.
  4. For each $j \in [\sigma], q \in [w_{m+1,j}/\ell]$, it then runs $\mathsf{pshare}(\overline{\mathbf{z}}^{j,q,\mathsf{left}}, D)$ and $\mathsf{pshare}(\overline{\mathbf{z}}^{j,q,\mathsf{right}}, D)$ to obtain shares $[\overline{\mathbf{z}}^{j,q,\mathsf{left}}]$ and $[\overline{\mathbf{z}}^{j,q,\mathsf{right}}]$ respectively. It also sends the respective shares to all parties.
- For each $j \in [\sigma], q \in [w_{m+1,j}/\ell]$, all parties locally subtract the randomness from these packed secret sharings as follows— $[\mathbf{z}^{j,q,\mathsf{left}}] = [\overline{\mathbf{z}}^{j,q,\mathsf{left}}] - [\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}] = [\overline{\mathbf{z}}^{j,q,\mathsf{right}}] - [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]$.

Output: The parties output their shares in $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m \in [d]$, $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$.

Fig. 6: A Protocol for Layer-wise Circuit Evaluation

6.2 Secure Layer-Wise Circuit Evaluation

This sub-protocol evaluates the circuit in a layer-wise fashion, i.e., it evaluate all gates in a given layer simultaneously. It takes properly aligned input vectors $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}]\right\}_{j\in[\sigma], q\in[w_{1,j}/\ell]}$ held by a set of parties, and computes packed shares $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m \in [d+1]$, $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$. We note that for notational convenience, this sub-protocol takes as input $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}], [\mathbf{x}_2^{j,q}], [\mathbf{y}_2^{j,q}]\right\}_{j\in[\sigma], q\in[w_{m,j}/\ell]}$

instead of just $\left\{[\mathbf{x}_1^{j,q}],[\mathbf{y}_1^{j,q}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$. This is because in our maliciously secure protocol, we invoke this sub-protocol for evaluating the circuit on actual inputs as well as on randomized inputs. When computing on actual inputs, we set $\mathbf{x}_1^{j,q}=\mathbf{x}_2^{j,q}$ and $\mathbf{y}_1^{j,q}=\mathbf{y}_2^{j,q}$ and when computing on randomized inputs, we set $\mathbf{x}_2^{j,q}=\mathbf{r}\mathbf{x}_1^{j,q}$ and $\mathbf{y}_2^{j,q}=\mathbf{r}\mathbf{y}_1^{j,q}$. A detailed description of this sub-protocol appears in Figure 6.

7 Our Order-C Semi-Honest Protocol

In this section, we describe our semi-honest protocol. All parties get a finite field $\mathbb{F}$ and a layered arithmetic circuit C (of width w and no. of gates $|\mathsf{C}|$) over $\mathbb{F}$ that computes the function f on inputs of length n as auxiliary inputs.[8]

Protocol: For each $i\in[n]$, party P_i holds input $x_i\in\mathbb{F}$ and the protocol proceeds as follows:

1. **Input Sharing Phase:** All the parties $\{P_1,\ldots,P_n\}$ collectively invoke $f_{\mathsf{pack-input}}$ as follows – every party P_i for $i\in[n]$, sends each of its inputs to the functionality $f_{\mathsf{pack-input}}$ and records its vector of packed shares $\left\{[\mathbf{x}^{j,q}],[\mathbf{y}^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ of the inputs as received from $f_{\mathsf{pack-input}}$. They set $[\mathbf{z}_1^{j,q,\mathsf{left}}]=[\mathbf{x}^{j,q}]$ and $[\mathbf{z}_1^{j,q,\mathsf{right}}]=[\mathbf{y}^{j,q}]$ for each $j\in[\sigma]$ and $q\in[w_{1,j}/\ell]$.
2. **Circuit Evaluation:** The parties collectively run sub-protocol π_{eval} on input shares $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}],[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$.
3. **Output Reconstruction:** For each $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}],[\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d+1,j}/\ell]}$, the parties run the reconstruction algorithm of packed secret sharing to learn the ouput.

We prove semi-honest security of this protocol in Section D. Next we calculate the complexity of this protocol.

Complexity of Our Semi-Honest Protocol. For each layer in the protocol, we generate $5\times$ (width of the layer/ℓ) packed shares, where $\ell=n/\epsilon$. We have $t=n\left(\frac{1}{2}-\frac{2}{\epsilon}\right)$. In the semi-honest setting, $n-t=n(\frac{1}{2}+\frac{2}{\epsilon})$ of these can be computed with n^2 communication. Therefore, overall the total communication required to generate all the correlated random packed shares is $5\times|\mathsf{C}|2\epsilon^2/(4+\epsilon)=10|\mathsf{C}|\epsilon^2/(4+\epsilon)$.

Additional communication required to evaluate each layer of the circuit is $5n\times$ (width of the layer/ℓ). Therefore, overall the total communication to generate correlated randomness and to evaluate the circuit is $10|\mathsf{C}|\epsilon^2/(4+\epsilon)+5|\mathsf{C}|\epsilon=\frac{5|\mathsf{C}|\epsilon(3\epsilon+4)}{4+\epsilon}$. An additional overhead to generate packed input shares for all inputs is at most $4n|\mathcal{I}|$, where $|\mathcal{I}|$ is the number of inputs to the protocol. Therefore, the total communication complexity is $\frac{5|\mathsf{C}|\epsilon(3\epsilon+4)}{4+\epsilon}+4n|\mathcal{I}|$.

8 Our Order-C Maliciously Secure Protocol

In this section, we present our maliciously secure Order-C MPC protocols. In addition to the sub protocols/functionalities from Sections 5 and 6, this construction also depends on some additional sub-protocols/functionalities. In this section, we first present those additional sub-protocols and then proceed to describe our maliciously secure order C MPC protocol.

8.1 Generating Random Packed Shares

In this section we describe a natural extension to π_{rand} that allows for generation of random *packed* shares. In our malicious secure protocol, these random values will allow us to efficiently check for linear errors injected by the adversary. We actually require two slightly different functionalities that we will represent in a single ideal functionality, as they are deeply related. The first functionality will generate packed sharings of vectors

[8] For simplicity we assume that each party has only one input. But our protocol can be trivially extended to accommodate scenarios where each party has multiple inputs.

in which each element is independent. The second functionality will generate packed sharing of vectors in which each element is the same random value. The ideal functionality $f_{\mathsf{pack-rand}}$ is described in Figure 7. Since the adversary can choose its own shares in the protocol, similar to Chida et. all [CGH+18], we let adversary send shares of the corrupted parties to the ideal functionality. The protocol realizing this ideal functionality is given in Section E.1.

The functionality $f_{\mathsf{pack-rand}}(\{P_1, \ldots, P_n\})$

The n-party functionality $f_{\mathsf{pack-rand}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

- Each honest party sends $\mathsf{mode} \in \{\mathsf{independent}, \mathsf{uniform}\}$ to the ideal functionality. If the honest players do not agree, the ideal functionality outputs $\perp$ and aborts.
- The ideal simulator Sim sends U_i for each corrupt party $i \in \mathcal{A}$.
- If $\mathsf{mode} = \mathsf{independant}$, the functionality $f_{\mathsf{pack-rand}}$ chooses a random vector $R \in \mathbb{F}^{\ell}$ such that each value in $r \in R$ is sampled independently from the field.
- If $\mathsf{mode} = \mathsf{uniform}$, the functionality $f_{\mathsf{pack-rand}}$ chooses a random value $r \in \mathbb{F}$ and sets $R \in \mathbb{F}^{\ell}$ to be r repeated ℓ times.
- The functionality $f_{\mathsf{pack-rand}}$ sets $[R]_{\mathcal{A}} = \{U_i\}_{i \in \mathcal{A}}$. It runs $\mathsf{pshare}(R, \mathcal{A}, [R]_{\mathcal{A}}, T)$ to receive a share R_i for each party P_i.
- It hands each honest party P_j its share R_j.

Fig. 7: Packed random share generation functionality

8.2 Checking Equality to Zero

In this section, we discuss the protocol of Chida et.al [CGH+18] to check whether a given sharing is a sharing of the value 0, without revealing any further information on the shared value. We extend this protocol to consider packed secret shares with the natural definition. We describe the protocol in Section E.2 and functionality realized by this protocol in Figure 8.

The functionality $f_{\mathsf{checkZero}}(\{P_1, \ldots, P_n\})$

The n-party functionality $f_{\mathsf{checkZero}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim receives $[\mathbf{v}]_{\mathcal{H}}$ from the honest parties and uses them to compute $\mathbf{v}$.

- If $\mathbf{v} = 0^{\ell}$, then $f_{\mathsf{checkZero}}$ sends 0 to the ideal adversary Sim. If Sim responds with reject (resp., accept), then $f_{\mathsf{checkZero}}$ sends reject (resp., accept) to the honest parties.
- If $\mathbf{v} \neq 0^{\ell}$, then $f_{\mathsf{checkZero}}$ proceeds as follows:
  - With probability $\frac{1}{|\mathbb{F}|}$ it sends accept to the honest parties and ideal adversary Sim.
  - With probability $1 - \frac{1}{|\mathbb{F}|}$ it sends reject to the honest parties and ideal adversary Sim.

Fig. 8: Random share generation functionality

8.3 Secure Dual Evaluation upto Linear Attacks

Next, we define a subprotocol, which takes packed shares of the actual inputs of the parties and packed shares of the randomized inputs and performs a dual evaluation of the circuit on these sets of inputs. Looking ahead, in our main protocol, this sub-protocol will be directly used for circuit evaluation and for maintaining the invariant that for each intermediate packed shared vector $\mathbf{z}$, the parties also compute shares of both $r\mathbf{z}$.

In [GIP15], Genkin et al. had shown that most packed secret sharing based semi-honest protocols satisfy the following property – when run in the presence of a malicious adversary, any attack strategy of the adversary is limited to simply injecting linear attacks on the outputs of multiplication gates. More precisely, a linear attack on multiplication gates is defined by an arbitrary linear combination of the vectors input to the set of gates.

Definition 4 (Linear Attack). *When multiplying two vectors $\mathbf{a}, \mathbf{b}$ of length ℓ each, a linear attack $\mathbf{L} = (\mathbf{L}_{\mathsf{left}}, \mathbf{L}_{\mathsf{right}})$ specifies linear functions $f_{\mathsf{left}} : \mathbb{F}^\ell \to \mathbb{F}^\ell$ and $f_{\mathsf{right}} : \mathbb{F}^\ell \to \mathbb{F}^\ell$, such that the output vector $\mathbf{c}$ is equal to $\mathbf{c} = \mathbf{a} \odot \mathbf{b} + f_{\mathsf{left}}(\mathbf{a}) + f_{\mathsf{right}}(\mathbf{b})$, where $\odot$ denotes the point-wise multiplication of two vectors.*

An important point to note about these attacks is that the linear attack $\mathbf{L}_{\mathsf{left}}$ is determined based on how $\mathbf{b}$ was secret shared and linear attack $\mathbf{L}_{\mathsf{right}}$ is determined based on how $\mathbf{a}$ was secret shared.

We observe that similar to most semi-honest protocols based on packed secret sharing, our sub-protocol for dual circuit evaluation also has the property that in the presence of a malicious adversary, any attack strategy of the adversary is limited to simply injecting linear attacks on to the outputs of each gate. We now give a description of this sub-protocol $\pi_{\mathsf{dual-eval}}$ and later give a proof sketch for this protocol in Section E.4.

Inputs: The parties $\{P_i\}_{i\in[n]}$ hold packed secret sharings $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{y}_1^{j,q,\mathsf{right}}], [\mathbf{rz}_1^{j,q,\mathsf{left}}], [\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$.
Dual Circuit Evaluation: The parties collectively run 2 executions of a truncated version of π_{eval} on inputs $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}], [\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$, and $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}], [\mathbf{rz}_1^{j,q,\mathsf{left}}], [\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ respectively, where in the layer $m = d$, the leader locally computes the masked output vectors, but does not secret share it among the other parties.
Output: The parties output their shares in $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m \in [d-1]$, $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$. P_{leader} outputs $\{\overline{\mathbf{z}}^{j,q,\mathsf{left}}, \overline{\mathbf{z}}^{j,q,\mathsf{right}}\}_{j\in[\sigma],q\in[w_{d+1,j}]}$.

8.4 Secure Multiplication upto Linear Attacks

In this section we describe a semi-honest secure multiplication protocol for packed secret shares that is secure up to linear attacks. We require this functionality to realize $f_{\mathsf{checkZero}}$ for packed secret sharing and setting up the randomized protocol execution for malicious security. The ideal functionality for $f_{\mathsf{pack-mult}}$ is given in Figure 9 and the protocol is given in Section E.3.

8.5 Maliciously Secure Protocol

We now describe a protocol that achieves security with abort against malicious corruptions.
Auxiliary Inputs: A finite field $\mathbb{F}$ and a layered arithmetic circuit C (of width w and $|\mathsf{C}|$ gates) over $\mathbb{F}$ that computes the function f on inputs of length n.
Inputs: For each $i \in [n]$, party P_i holds input $x_i \in \mathbb{F}$.
Protocol: (Throughout the protocol, if any party receives $\perp$ as output from a call to a sub-functionality, then it sends $\perp$ to all other parties, outputs $\perp$ and halts):

1. **Secret-Sharing Inputs:** All the parties $\{P_1, \ldots, P_n\}$ collectively invoke $f_{\mathsf{pack-input}}$ as follows — every party P_i for $i \in [n]$, sends each of its input x_i to functionality $f_{\mathsf{pack-input}}$. and records its vector of packed shares $\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ of the inputs as received from $f_{\mathsf{pack-input}}$. They set $[\mathbf{z}_1^{j,q,\mathsf{left}}] = [\mathbf{x}^{j,q}]$ and $[\mathbf{z}_1^{j,q,\mathsf{right}}] = [\mathbf{y}^{j,q}]$ for each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$.

The functionality $f_{\mathsf{pack-mult}}(\{P_1,\ldots,P_n\})$

The functionality $f_{\mathsf{pack-mult}}$, running with a set of parties $\{P_1,\ldots,P_n\}$ and the ideal adversary Sim proceeds as follows:

- Upon receiving $[\mathbf{x}]_{\mathcal{H}}$ and $[\mathbf{y}]_{\mathcal{H}}$ from the honest parties, the ideal functionality $f_{\mathsf{pack-mult}}$ reconstructs computes $\mathbf{x}=\{x_i\}_{i\in[\ell]}, \mathbf{y}=\{y_i\}_{i\in[\ell]}$. The simulator also computes shares $[\mathbf{x}]_{\mathcal{A}}$ and $[\mathbf{y}]_{\mathcal{A}}$ and sends them to the adversary.
- Upon receiving Linear error $\mathbf{L}:\mathbb{F}^{2\ell}\rightarrow\mathbb{F}^{\ell}$ and $\{\mathbf{u}_i\}_{i\in\mathcal{A}}$ from the ideal adversary Sim, functionality $f_{\mathsf{pack-mult}}$ defines $\mathbf{z}=\mathbf{x}\cdot\mathbf{y}+\mathbf{L}(\mathbf{x},\mathbf{y})$ and $[\mathbf{z}]_{\mathcal{A}}=\{\mathbf{u}_i\}_{i\in\mathcal{A}}$. It then runs $\mathsf{pshare}(\mathbf{z},\mathcal{A},[\mathbf{z}]_{\mathcal{A}},D)$ to obtain a share $\mathbf{z}_j$ for each party P_j.
- The ideal functionality $f_{\mathsf{pack-mult}}$ hands each honest party P_j its share $\mathbf{z}_j$.

Fig. 9: Secure Multiplication Up to Linear Attack functionality

2. **Pre-processing:**
   - Random Input Generation: The parties invoke $f_{\mathsf{pack-rand}}$ on mode $\mathsf{uniform}$ to receive packed sharings $[\mathbf{r}]$ of a vector $\mathbf{r}$, of the form $\mathbf{r}=(r,\ldots,r)$.
   - The parties also invoke $f_{\mathsf{pack-rand}}$ on mode $\mathsf{independent}$ to receive packed sharings $\{[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}],[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}]\}_{m\in[d],j\in[\sigma],q\in[w_{m,j}/\ell]}$ of random vectors $\boldsymbol{\alpha}_m^{j,q,\mathsf{left}},\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}$.
   - Randomizing Inputs: For each packed input sharing $[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}]$ (for $j\in[\sigma],q\in[w_{1,j}/\ell]$), the parties invoke f_{mult}, on $[\mathbf{z}_1^{j,q,\mathsf{right}}]$ and $[\mathbf{r}]$ to receive $[\mathbf{rz}_1^{j,q,\mathsf{left}}]$ and on $[\mathbf{z}_1^{j,q,\mathsf{right}}]$ and $[\mathbf{r}]$ to receive $[\mathbf{rz}_1^{j,q,\mathsf{right}}]$.
3. **Dual Circuit Evaluation:** The parties run $\pi_{\mathsf{dual-eval}}$ on inputs $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{y}_1^{j,q,\mathsf{right}}],[\mathbf{rz}_1^{j,q,\mathsf{left}}],[\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ to obtain shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}],[\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ and $\left\{[\mathbf{rz}_{m+1}^{j,q,\mathsf{left}}],[\mathbf{rz}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ for each $m\in[d-1]$ and the leader party additionally receives the masked output vectors for the last layer. The parties then compute the last two-steps of π_{eval} for the last layer. i.e., the leader party then pack secret shares these vectors among the other parties and all the parties subtract their shares of the random masks from these packed secret shares to obtain shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}],[\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d,j}/\ell]}$ and $\left\{[\mathbf{rz}_{d+1}^{j,q,\mathsf{left}}],[\mathbf{rz}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d,j}/\ell]}$.
4. **Verification Step:** Each party does the following:
   (a) For each $m\in[d+1]$, $j\in[\sigma]$,$q\in[w_{m,j}/\ell]$, the parties invoke f_{mult} on their packed shares $([\mathbf{z}_m^{j,q,\mathsf{left}}]$, $[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}])$, $([\mathbf{rz}_m^{j,q,\mathsf{left}}]$, $[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}])$, $([\mathbf{z}_m^{j,q,\mathsf{right}}]$, $[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}])$ and $([\mathbf{rz}_m^{j,q,\mathsf{right}}]$, $[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}])$, and locally compute. [9]

$$[\mathbf{v}]=\sum_{m\in[d+1]}\sum_{j\in[\sigma],q\in[w_{m,j}/\ell]}[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}][\mathbf{rz}_m^{j,q,\mathsf{left}}]+[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}][\mathbf{rz}_m^{j,q,\mathsf{right}}]$$

$$[\mathbf{u}]=\sum_{m\in[d+1]}\sum_{j\in[\sigma],q\in[w_{m,j}/\ell]}[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}][\mathbf{z}_m^{j,q,\mathsf{left}}]+[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}][\mathbf{z}_m^{j,q,\mathsf{right}}]$$

   (b) The parties open shares $[\mathbf{r}]$ to reconstruct $\mathbf{r}=(r,\ldots,r)$.

[9] We remark that for notational convinience we describe this step as consisting of $4|\mathsf{C}|/\ell$ multiplications (and hence these many degree reduction steps), it can be done with just two degree reduction step, where the parties first locally multiply and add their respective shares to compute $\langle\mathbf{v}\rangle$ and $\langle\mathbf{u}\rangle$ and then communicate to obtain shares of $[\mathbf{v}]$ and $[\mathbf{u}]$ respectively.

(c) Each party then locally computes $[\mathbf{t}] = [\mathbf{v}] - r[\mathbf{u}]$
(d) The parties invoke $f_{\mathsf{checkZero}}$ on $[\mathbf{t}]$. If $f_{\mathsf{checkZero}}$ outputs reject, the output of the parties is $\perp$. Else, if it outputs accept, then the parties proceed.

5. **Output Reconstruction:** For each output vector, the parties run the reconstruction algorithm of packed secret sharing to learn the output. If the reconstruction algorithm outputs $\perp$, then the honest parties output $\perp$ and halt.

The proof of security for this protocol is given in Section F. We note that the above protocol only works for circuits over large arithmetic fields. In Section F, we also present an extension to a protocol that works for circuits over smaller fields.

Complexity Calculation for our Maliciously Secure Protocol over Large Fields. For each layer in the protocol, we generate $5\times$ (width of the layer/ℓ), where $\ell = n/\epsilon$. We have $t = n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$. In the malicious setting, $n-t \approx n(\frac{1}{2}+\frac{2}{\epsilon})$ of these packed shares can be computed with $5n^2$ communication. Therefore, overall the total communication required to generate all the randomness is the following:

— Correlated randomness for evaluating the circuit on actual inputs: $\frac{|C|}{\frac{n}{\epsilon}\times n(\frac{1}{2}+\frac{2}{\epsilon})}5n^2 = \frac{10\epsilon^2|\mathsf{C}|}{\epsilon+4}$.

— Correlated randomness for evaluating the circuit on randomized inputs: $\frac{10\epsilon^2|\mathsf{C}|}{\epsilon+4}$.

— Shares of random $\boldsymbol{\alpha}$ vectors: $\frac{2\epsilon|\mathsf{C}|(3\epsilon-4)}{\epsilon+4}$

Additional communication required for dual execution of the circuit is $2\times 5\times n\times$ (width of the layer/ℓ). Therefore, overall the total communication to generate correlated randomness and for the dual evaluate the circuit is $\frac{(26\epsilon^2-8\epsilon)|\mathsf{C}|}{\epsilon+4} + 10|\mathsf{C}|\epsilon = \frac{36\epsilon^2|\mathsf{C}|+32\epsilon|\mathsf{C}|}{\epsilon+4}$. An additional overhead to generate packed input shares for all inputs is $n^2|\mathcal{I}|$, where $|\mathcal{I}|$ is the number of inputs to the protocol. The communication required to generate shares of randomized inputs is $n^2|\mathcal{I}|$. Finally, the verfication step only requires $2n^2$ communication. Therefore, the total communication complexity is $\frac{36\epsilon^2|\mathsf{C}|+32\epsilon|\mathsf{C}|}{\epsilon+4} + 2n^2|\mathcal{I}|$.

9 Implementation and Evaluation

In this section, we present the details of our implementation and do a detailed comparison with prior work.

9.1 Comparison

We start by comparing the concrete efficiency of our protocol based on the calculations from Section 8.5, where we show that the total communication complexity of our maliciously secure protocol is $\frac{36\epsilon^2|\mathsf{C}|+32\epsilon|\mathsf{C}|}{\epsilon+4} + 2n^2|\mathcal{I}|$. Recall that our protocol achieves security against $t < n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$ corruptions; we do our comparison with the state-of-the-art using the same corruption threshold as they consider.

The state-of-the-art in this regime is the $O(n|\mathsf{C}|)$ protocol of [FL19] for $t < n/3$ corruptions, that requires each party to communicate approximately $4\frac{2}{3}$ field elements per multiplication gate. In contrast, for $n = 125$ parties and $t < n/3$ corruptions, our protocol requires each party to send approximately $2\frac{3}{4}$ field elements per gate, in expectation. Notice that while we require parties to communicate for every gate in the circuit, [FL19] only requires communication per multiplication gate. However, it is easy to see that for circuits with approximately 65% multiplication gates, our protocol is expected (in theory) to outperform [FL19] for 125 parties.

As discussed earlier, a nice advantage of $O(|\mathsf{C}|)$ protocols is that the per-party communication in these protocols goes down as the number of parties increases. For instance, for the same corruption threshold of $t < n/3$, and $n = 150$ parties, our protocol would (on paper) only require each party to communicate $2\frac{1}{3}$ field elements per gate. In this case, our protocol is already expected to perform better than [FL19] for circuits that have more that 55% multiplication gates. In fact, as the number of parties increase, the percentage of the circuit that must be multiplication gates in order to show improvements reduces.

Table 2: Comparing the runtime of our protocol and that of related work. Results are reported for the average protocol execution time over five randomized circuits each with 1,000,000 gates for WAN. LAN results are for a single execution with 1,000,000 gates. All times are in milliseconds and n is the number of parties. Asterisk denote estimated runtimes where data was missing (see text).

Paper	This Work	This Work	[CGH+18]	[FL19]	This Work	This Work	[CGH+18]	[CGH+18]	[CGH+18]
Network Config	LAN				WAN				
t	$n/4$	$n/3$	$n/2$	$n/3$	$n/4$	$n/3$	$n/2$	$n/2$	$n/2$
Depth	1,000	1,000	1,000	20	1,000	1,000	20	100	1,000
$n = 30$	28,709	29,014	12,144	1,241	261,029	298,520	87,355	34,860	376,464*
$n = 50$	36,544	40,537	26,310	1,891	206,475	285,074	128,366	197,321	815,610*
$n = 70$	48,729	54,692	33,294	2,585	186,535	214,575	164,145*	251,286*	1,032,114*
$n = 90$	52,563	54,477	48,927	3,689	278,038	260,995	204,166*	355,167*	1,516,737*
$n = 110$	60,281	62,871	79,728	$> 3,999$	270,558	305,071	256,711*	478,361*	2,471,568*
$n = 150$	-	-	-	-	282,381	315,182	-	-	-
$n = 200$	-	-	-	-	262,621	279,111	-	-	-
$n = 250$	-	-	-	-	301,555	320,477	-	-	-
$n = 300$	-	-	-	-	335,588	378,262	-	-	-

Since the communication complexity of our protocol is dependent on the tunable parameter ϵ (that is directly proportional to the corruption threshold t), the efficiency of our protocol is expected to increase further even for fewer parties, if we allow for lower corruption threshold. For instance, for $t < n/4$ corruptions and $n = 100$ parties, we require per-party communication of $2\frac{1}{7}$ field elements per gate.

Finally, we remark that, the above is only a theoretical comparison. Indeed the complexity calculation in Section 8.5 was done assuming the "best-case scenario", e.g., where the circuit is such that it has exactly $n-t$ repetitions of the same kinds of blocks, and that each block has an exact multiple of n/ϵ gates and n is exactly divisible by ϵ etc. In practice, this may not be the case. When the circuit is not perfectly divisible, there may be some "waste," meaning either more randomness will be generated than is needed or some packed secret sharings will not be completely filled. The effects of this waste can be seen in our implementation below, where the runtime may even decrease slightly (see $t = n/3$ for 70 and 90 parties) as the number of parties increases because the division is more efficient.

To make our comparison more concrete, we implement our protocol and evaluate it on different network settings. While we do not get the exact same improvements as derived above (likely due to waste), we clearly demonstrate that our protocol is practical for even small numbers of parties, and becomes more efficient than state-of-the-art for large numbers of parties.

9.2 Implementation

We implemented our maliciously secure protocol from Section 8.5. Our implementation is in `C++` and uses libscapi [Cry19] to provide communication and circuit parsing. Since this library does not support packed secret sharing or the non-interactive packing of traditional secret shares (Section 5.3), we implement them within the context of the library. Our protocol implementation automatically generates batches of correlated randomness on the fly as needed. During circuit evaluation, gates within each block are divided into packs according to the number of players and the packing constant. Randomness is then retrieved from a pool; if no suitable randomness is available from a previous execution of the randomness generation subprotocol, the players pause to generate a fresh batch of randomness and verify that it is correct. This reduces the need to set aside large amounts of memory at the beginning of computation.

To evaluate our implementation, we generate layered circuits that satisfy the highly repetitive structural requirements. Specifically, we generate a fixed number of layers of constant depth, each containing addition

and multiplication gates that are randomly wired together. These layers are then repeated as a group some fixed number of times. Benchmarking on random circuits is common, accepted practice for honest majority protocols [CGH+18,FL19]. We modify the circuit format from a standard one used by libscapi [Cry19] to help make this representation more succinct. Specifically, because our protocol only operates over layered circuits, we have gates take in wires indexed relatively from the previous layer, instead of using global indices. Additionally, because layers are repeated many times, we just indicate the order of layers, rather than writing out the layers explicitly.

We ran tests in two network deployments, the first to measure the performance independent of network delay and the second to measure the effect of network communication. In our first deployment, we ran all of the parties on a single, large server with two Intel(R) Xeon(R) CPU E5-2695 @ 2.10 GHz. In our second deployment, parties were split evenly across three different AWS regions: us-east-1, us-east-2, and us-west-2. Each party was a separate c4.xlarge instance with 7.5 GB of RAM and a 2.9 GHz Intel Xeon E5-2666 v3 Processor.

We compare our work to the most efficient $O(n|\mathsf{C}|)$ work, as there is no comparable work which has been run for a large number of parties.[10] These works only run their protocol for up to 110 parties. Therefore our emphasis is not on direct time result comparisons, but instead on relative efficiency even with small numbers of players. We note that the protocols against which we compare do not require highly repetitive circuits; while this might make it seem like we are performing an apple-to-oranges comparison, there is no efficiency gain for running these $O(n|\mathsf{C}|)$ protocols over highly repetitive circuits. As such, the timing results that these works present would hold true for highly repetitive circuits as well, and our comparison is apples-to-apples for highly repetitive circuits (up to the percentage of addition gates, which we discussed above.)

We compare the runtime of our protocol in both our LAN deployment and WAN deployment to [CGH+18,FL19] in Table 2. Because of differences between our protocol and intended applications, there are several important things to note in this comparison. First, we run all our tests on circuits with depth 1,000 to ensure there is sufficient repetition in the circuit. Furukawa et al. use only a depth 20 circuit in their LAN tests, meaning more parallelism can be leveraged. We note that when Chida et al. increase the depth of their circuits from 20 to 1,000 in their LAN deployment, the runtime for large numbers of parties increases 5-10x [CGH+18]. If we assume [FL19] will act similarly, we see that their runtime is approximately half of ours, when run with small number of parties. This is consistent with their finding that their protocol is about twice as fast as [CGH+18]. We emphasise that for larger numbers of parties our protocol is expected to perform better.

Because Chida et al. only run their protocol for up to 30 players and up to circuit depth 100 in their WAN deployment, there is missing data for our comparison. We note that their WAN runtimes are consistently just over 30x higher than their LAN deployment. Using this observation, we extrapolate estimated runtimes for their protocol under different configurations, denoted with an asterisk. We emphasise that this estimation is rough, and all these measurements should be interpreted with a degree of skepticism; we include them only to attempt a more consistent comparison to illustrate the general trends of our preliminary implementation.

Our results show that our protocol, even using an un-optimized implementation, is comparable to these works for small numbers of parties (see Table 2). For larger numbers of parties (see Table 2), where we have no comparable results, there is an upward trend in protocol execution time. This could be a result of networking overhead or varying levels of network congestion when each of the experiments was performed. For example, when executing with 250 parties and a corruption threshold of n/4 the difference between the fastest and slowest execution time was over 60,000 ms, whereas in other deployments the difference is as low as 1,000 ms. In general, an increase is also expected as asymptotic complexity has an additive quadratic dependency on n with the input size of the circuit. Overall our experiments demonstrate that our protocol does not introduce an impractical overhead in its effort to achieve $O(|\mathsf{C}|)$ MPC. *As the number of parties continues to grow (*e.g. *hundreds or thousands), the benefits of our protocol will become even more apparent.*

[10] The only protocol to be run on large numbers of parties rests on incomparable assumptions like CRS [WJS+19].

10 Acknowledgements

The first and second authors are supported in part by NSF under awards CNS-1653110 and CNS-1801479 and the Office of Naval Research under contract N00014-19-1-2292. The first author is also supported in part by DARPA under Contract No. HR001120C0084. The second and third authors are supported in part by an NSF CNS grant 1814919, NSF CAREER award 1942789 and Johns Hopkins University Catalyst award. The third author is additionally partly supported by Office off Naval Research grant N00014-19-1-2294. The forth author is supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project. The forth author would like to thank Mayank Varia for suggesting machine learning as an application of highly repetitive circuits. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

AHIV17. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

BR93. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

BTH08. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Heidelberg, March 2008.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, August 1988.

CDG+17. Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

CGH+18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

Cry19. Cryptobiu. cryptobiu/libscapi, May 2019.

DI06. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, August 2006.

DIK+08. Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261. Springer, Heidelberg, August 2008.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg, May / June 2010.

DMS04. Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

FL19. Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1557–1571. ACM Press, November 2019.

FS87. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

FY92. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.

Gen16. Daniel Genkin. *Secure Computation in Hostile Environments*. PhD thesis, Technion - Israel Institute of Technology, 2016.

GFD09. Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), June 2009. U.S.Department of Commerce/National Institute of Standards and Technology.

GIP+14. Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

GIP15. Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015.

GMO16. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

GMR85. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

Gol04. Oded Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

GSB+16. Adria Gascon, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Secure linear regression on vertically partitioned datasets. Cryptology ePrint Archive, Report 2016/892, 2016. http://eprint.iacr.org/2016/892.

GSY21. S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale mpc. Springer-Verlag, 2021.

GSZ20. Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 618–646. Springer, Heidelberg, August 2020.

HHNZ19. Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

HN06. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 463–482. Springer, Heidelberg, August 2006.

IKOS07. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017.

MGC+16. Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

MZ17. Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

Nak08. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. 2008.

NIS02. Fips pub 180-2, secure hash standard (shs), 2002. U.S.Department of Commerce/National Institute of Standards and Technology.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 321–339. Springer, Heidelberg, July 2018.

RSG98. M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

Sha79a. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

Sha79b. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

WJS+19. Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, November 2019.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

A Preliminaries (contd.)

In this section, we present standard definitions of secure multiparty computation, regular threshold secret sharing (with a special requirement) and packed secret sharing.

A.1 Secure Multiparty Computation

A secure multi-party computation protocol (MPC) is a protocol executed by n parties $\mathcal{P} = \{P_1, \cdots, P_n\}$ for a functionality $\mathcal{F}$. We allow for parties to exchange messages simultaneously. In every round, every party is allowed to exchange messages with other parties using different communication channels, depending on the model. A protocol is said to have k *rounds* if it proceeds in k distinct and interactive rounds.

Adversarial Behavior One of the primary goals in MPC is to protect the honest parties against dishonest behavior of the corrupted parties. This is usually modeled using a central adversarial entity, that controls the set of corrupted parties and instructs them on how to operate. That is, the adversary obtains the views of the corrupted parties, consisting of their inputs, random tapes and incoming messages, and provides them with the messages that they are to send in the execution of the protocol. In our protocols we only consider the case where the adversary can only control a minority of the parties in the protocol. We discuss the following adversarial models in detail:

1. **Semi-Honest Adversaries:** A semi-honest adversary always follows the instructions of the protocol. This is an "honest but curious" adversarial model, where the adversary might try to learn extra information by analyzing the transcript of the protocol later.
2. **Malicious Adversaries:** A malicious adversary can deviate from the protocol and instruct the corrupted parties to follow any arbitrary strategy.

We provide the basic definitions for secure multiparty computation according to the real/ideal paradigm [Gol04]. Informally, a protocol is considered secure if whatever an adversary can do in the real execution of protocol, can be done also in an ideal computation, in which an uncorrupted trusted party assists the computation.

Security Definitions Real World. The real world execution of a protocol $\mathit{\Pi} = (P_1, \ldots, P_n)$ begins by an adversary $\mathcal{A}$ selecting any arbitrary subset of parties $\mathcal{I} \subset [n]$ to corrupt. The parties then engage in an execution of a real n-party protocol $\mathit{\Pi}$. Throughout the execution of $\mathit{\Pi}$, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of $\mathit{\Pi}$. At the conclusion of the protocol, each honest party outputs all the outputs it obtained in the computations. Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$. This joint execution of $\mathit{\Pi}$ under $(\mathcal{A}, \mathcal{I})$ in the real model, on input vector $\mathbf{x} = (x_1, \ldots, x_n)$, auxiliary input z and security parameter λ, denoted by $\mathsf{REAL}_{\mathit{\Pi},\mathcal{I},\mathcal{A}(z)}\left(1^\lambda, \mathbf{x}\right)$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}(z)$ resulting from this protocol interaction.

Ideal World. We now present standard definitions of ideal-model computations that are used to define security with abort. We start by presenting the ideal-model computation for security with abort, where the adversary may abort the computation either before or after it has learned the output; other ideal-model computations are defined either by allowing the adversary to selectively abort to some parties but not to others or by restricting the power of the adversary either by forcing the adversary to identify a corrupted party in case of abort, or no abort (guaranteed output delivery).

Ideal Computation with Abort. An ideal computation with abort of an n-party functionality $\mathcal{F}$ on input $\mathbf{x} = (x_1, \ldots, x_n)$ for parties $(P_1, \ldots, P_n)$ in the presence of an ideal-model adversary $\mathcal{A}$ controlling the parties indexed by $\mathcal{I} \subset [n]$, proceeds via the following steps.

Sending inputs to trusted party: For each $i \notin \mathcal{I}$, P_i sends its input x_i to the trusted party. If $i \in \mathcal{I}$, the adversary may send to the trusted party any arbitrary input for the corrupted party P_i. Let x_i' be the value actually sent as the ith party's input.

Early abort: The adversary $\mathcal{A}$ can abort the computation by sending an abort message to the trusted party. In case of such an abort, the trusted party sends $\perp$ to all parties and halts.

Trusted party answers adversary: The trusted party computes $(y_1, \ldots, y_n) = \mathcal{F}(x_1', \ldots, x_n')$ and sends y_i to party P_i for every $i \in \mathcal{I}$.

Late abort: The adversary $\mathcal{A}$ can abort the computation (after seeing the outputs of corrupted parties) by sending an abort message to the trusted party. In case of such abort, the trusted party sends $\perp$ to all honest parties and halts. Otherwise, the adversary sends a continue message to the trusted party.

Trusted party answers remaining parties: The trusted party sends y_i to P_i for every $i \notin \mathcal{I}$.

Outputs: Honest parties always output the message received from the trusted party and the corrupted parties output nothing. The adversary $\mathcal{A}$ outputs an arbitrary function of the initial inputs x_i s.t. $i \in \mathcal{I}$, the messages received by the corrupted parties from the trusted party and its auxiliary input.

Definition 5 (Ideal-model computation). *Let $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party functionality. let $\mathcal{I} \subset [n]$ be the set of indices of the corrupted parties, and let λ be the security parameter. Then, the joint execution of $\mathcal{F}$ under $(\mathcal{A}, \mathcal{I})$ in the ideal model, on input vector $\mathbf{x} = (x_1, \ldots, x_n)$, auxiliary input z to $\mathcal{A}$ and security parameter λ, denoted $\mathsf{IDEAL}_{\mathcal{F},\mathcal{I},\mathcal{A}(z)}(1^\lambda, \mathbf{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the above described ideal process.*

Security Having defined the real and ideal models, we can now define security of protocols according to the real/ideal paradigm. Since we work in the information-theoretic setting, we only give a definition for statistically secure protocols.

Definition 6. *Let $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party functionality and let Π be a probabilistic polynomial-time protocol computing $\mathcal{F}$. The protocol Π computes $\mathcal{F}$ with statistical security against at most t corruptions with abort, if for every unbounded real-model adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ for the ideal model, who's running time is polynomial in the running time of $\mathcal{A}$, such that for every $\mathcal{I} \subset [n]$ of size at most t, it holds that*

$$\left\{\mathsf{REAL}_{\Pi,\mathcal{I},\mathcal{A}(z)}\left(1^\lambda, \mathbf{x}\right)\right\}_{(\mathbf{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}} \approx_s \left\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{I},\mathcal{S}(z)}(1^\lambda, \mathbf{x})\right\}_{(\mathbf{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}}$$

A.2 Threshold Secret Sharing

A t-out-of-n secret sharing scheme enables n parties to share as secret $v \in \mathbb{F}$ so that no subset of t parties can learn any information about it, while any subset of $t+1$ parties can reconstruct it. We use *Shamir's secret sharing* scheme [Sha79b] in our protocols that supports the following procedures:

— $\mathsf{share}(v, t+\ell)$: In this procedure, a dealer shares a value $v \in \mathbb{F}$ as follows:
  1. Set $p_0 = v$ and sample a random polynomial $q(z)$ of degree t such that $q(v) = 0$.
  2. Set $p(z) = p_0 + q(z)\prod_{i=1}^{\ell}(z - e_i)$, where $e_1, \ldots, e_n$ are preselected elements in $\mathbb{F}$.
  3. For each $i \in [n]$, set $v_i = p(i)$.

  Each output share v_i (for $i \in [n]$) is the share intended for party P_i. We denote the $t+\ell$-out-of-n sharing of a value v by $[v]$. We use the notation $[v]_J$ to denote the shares held by a subset of parties $J \subset [n]$. We stress that if the dealer is corrupted, then the shares received by the parties may not be correct. Nevertheless, we abuse notation and say that the parties hold shares $[v]$ even if these are not correct.

— $\mathsf{share}(v, J, [v]_J, t+\ell)$: This procedure is similar to the previous procedure, except that here the shares of a subset J of parties with $|J| \le t$ are fixed in advance. Given the value v to be shared, let $p(z) = v + p_1 z + p_2 z^2 + \ldots + p_t z^{t+\ell}$ be the polynomial used for secret sharing. Now given $|J|$ shares, we get the following system of equations:

$$\forall i \in J,\ v_i = v + p_1 i + p_2 i^2 + \ldots + p_t i^t$$

  This a system of $|J|$ equations in t variables $\{p_1, \ldots, p_t\}$ and can be easily solved using Gaussian elimination. Finally, given the polynomial $p(z)$ the shares of all other parties $i \in [n] \setminus J$ is $v_i = p(i)$.

A.3 Threshold Packed Secret Sharing

A packed secret sharing scheme enables n parties to share a block of ℓ secrets $v = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ so that no subset of at most $t - \ell + 1$ parties can learn any information about it, while any subset of $D + 1$ parties can reconstruct it. We use the multi-secret generalization of Shamir's secret sharing scheme as introduced by Franklin et. al [FY92]. Let $\alpha_1, \ldots, \alpha_n$ and $e_1, \ldots, e_\ell$ be $n + \ell$ preselected elements in $\mathbb{F}$ that are known to all parties. This packed secret sharing scheme supports the following procedures:

— $\mathsf{pshare}(s, D)$: In this procedure, a dealer shares a block of ℓ secrets $s = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ using a random polynomial $p(z)$ of degree D over $\mathbb{F}$, subject to the constraint $p(e_i) = s_i$ for each $1 \leq i \leq \ell$. This is done as follows:
  1. Pick a random polynomial $q(z)$ of degree $D - \ell$.
  2. Set
  $$p(z) = q(z) \prod_{i=1}^{\ell} (x - e_i) + \sum_{i=1}^{\ell} s_i L_i(z),$$
  where $L_i(z)$ is the Lagrange polynomial $\frac{\Pi_{j \neq i}(x - e_j)}{\Pi_{j \neq i}(e_i - e_j)}$.
  3. For each $i \in [n]$, send $p(\alpha_i)$ to party P_i.

— $\mathsf{pshare}(v, J, [v]_J, D)$: This procedure is similar to the packed secret sharing procedure using a univariate polynomial, except that here the shares of a subset J of parties with $|J| \leq D$ are fixed in advance. Given a block of ρ values $v = (s_1, \ldots, s_\rho)$ to be shared, let $p(z) = p_0 + p_1 z + p_2 z^2 + \ldots + p_{t+\rho} z^{t+\rho}$ be the polynomial used for secret sharing. Now given $|J|$ shares and ρ secret values $(s_1, \ldots, s_\rho)$, we get the following system of equations:
$$\forall i \in J,\ v_i = p_0 + p_1 i + p_2 i^2 + \ldots + p_{t+\rho} i^{t+\rho}$$
$$\forall i \in [\rho],\ v_i = p_0 + p_1 \mu_i + p_2 \mu_i^2 + \ldots + p_{t+\rho} \mu_i^{t+\rho}$$
This a system of $|J| + \rho$ equations in $t + \rho + 1$ variables $\{p_0, \ldots, p_{t+\rho}\}$ and can be easily solved using Gaussian elimination. Finally, given the polynomial $p(z)$ the shares of all other parties $i \in [n] \setminus J$ is $v_i = p(i)$.

B Input Sharing Phase

In this section, we present the sub-protocols that are used in the input sharing phase of our main protocol.

B.1 A Protocol for Generating Random Shares

In this section, we describe the protocol π_{rand} that securely realizes the functionality f_{rand} (Figure 3). The protocol proceeds as follows:
Auxiliary Inputs Hyper-invertible matrix $\mathbf{H}_{n,n}$
Inputs: The parties do not have any inputs.
Protocol π_{rand}: The parties proceed as follows:

— Each party $\{P_i\}$ (for $i \in [n]$) chooses a random element $u_i \in \mathbb{F}$. It runs $\mathsf{share}(u_i, t + \ell)$ to receive shares $[u_i]$. For each $j \in [n]$, it party P_j, its share in $[u_j]$.

— Given shares $([u_1], \ldots [u_n])$, the parties compute
$$([r_1], \ldots, [r_n]) = \mathbf{H}_{n,n}^T \cdot ([u_1], \ldots, [u_n])$$

— Each party sends its shares in $[r_{n-t+1}], \ldots, [r_n]$ to all other parties. The parties locally run $\mathsf{open}([r_{n-t+1}]), \ldots, \mathsf{open}([r_n])$ to check if all the shares lie on the same degree $t + \ell$ polynomial and moreover that the polynomial is of the form $r_{t+\ell} + q(z) \prod_{j=1}^{\ell} (z - e_j)$, where $q(z)$ is a degree t polynomial. If this check succeeds, then the parties send "pass" to all other parties, else they send "fail".

— If all n parties output pass, then the parties output their shares in $[r_1], \ldots, [r_{n-t}]$, else they output $\perp$ and halt.

Output: The parties output $[r_1], \ldots [r_{n-t}]$ or $\perp$.

Lemma 2. *This protocol securely computes $n-t$ instantiations of f_{rand} with abort in the presence of malicious adversaries who controls t parties.*

The proof of this Lemma follows from [BTH08], hence we omit it here.

B.2 A Protocol for Secret Sharing of Inputs

In this section, we describe the protocol π_{input} that securely realizes the functionality f_{input} (Figure 4). The protocol proceeds as follows:
Inputs: Let $x_1, \ldots, x_M \in \mathbb{F}$ be the series of inputs, each x_i is held by some party P_j.
Protocol π_{input}: The parties proceed as follows:

— The parties $\{P_1, \ldots, P_n\}$ invoke f_{rand} M times to obtain sharings $[r_1], \ldots, [r_M]$.
— For each $i \in [M]$, all the parties send their shares in $[r_i]$ to party P_j, who owns the input. Party P_j runs $\mathsf{open}([r_i])$. If it receives $\perp$, then it sends $\perp$ to all parties, outputs `abort` and halts.
— For each $i \in [M]$, party P_j (who owns input x_i) sends $v_i = x_i - r_i$ to all other parties.
— All parties send $\overrightarrow{v} = (v_2, \ldots, v_M)$ to all other parties. If any party receives a different vector to its own, then it outputs $\perp$ and halts.
— For each $i \in [M]$, the parties compute $[x_i] = [r_i] + v_i$.

Output: The parties output $[x_1], \ldots, [x_M]$

Lemma 3. *This protocol securely computes f_{input} with abort in the f_{rand}-hybrid model in the presence of a malicious adversary who controls at most t parties.*

Proof. Let $\mathcal{A}$ be the real adversary. We construct a simulator Sim as follows. Sim receives $[r_i]_{\mathcal{A}}$ for each $i \in [M]$ that $\mathcal{A}$, sends to f_{rand} in the protocol. For each $i \in [M]$, it samples random $r_i \in \mathbb{F}$ and computes $[r_i] \leftarrow \mathsf{share}(r_i, \mathcal{A}, [r_i]_{\mathcal{A}}, t+\ell)$. Sim then simulates the honest parties in all $\mathsf{reconstruct}$ executions. If an honest party P_j receives $\perp$ in the reconstruction, then Sim simulates it sending $\perp$ to all parties. Sim simulates the remainder of the execution, obtaining all v_i values from $\mathcal{A}$ associated with the corrupted parties' inputs, and sending random v_j values for inputs associated with honest parties. For every i for which the ith input is that of a corrupted party P_i, Sim sends $x_i = v_i + r_i$ to the ideal functionality f_{input}. For every $i \in [n]$, Sim defines the corrupted parties' shares $[x_i]_{\mathcal{A}}$ to be $[r_i + v_i]_{\mathcal{A}}$. Then Sim sends $[x_i]_{\mathcal{A}}, \ldots, [x_n]_{\mathcal{A}}$ to the ideal functionality f_{input}. For every honest party, if it aborted in the simulation, then Sim sends `abort` to the ideal functionality f_{input}, else, it sends continue. Finally Sim outputs whatever $\mathcal{A}$ outputs. While indistinguishability of the honest parties output follows trivially, indistinguishability of the corrupt parties' view in the real and ideal worlds follows from the fact that the adversary only gets to see t shares of the honest parties' inputs. From the privacy property of secret sharing, we know that t shares are not sufficient for reconstructing shares corresponding to a $t + \ell$ degree polynomial.

B.3 A Protocol for Packed Secret Sharing of Inputs

In this section, we describe the protocol $\pi_{\mathsf{pack-input}}$ that securely realizes the functionality $f_{\mathsf{pack-input}}$ (Figure 5). The protocol proceeds as follows:
Inputs: Let $x_1, \ldots, x_M \in \mathbb{F}$ be the series of inputs, each x_i is held by some party P_j.
Protocol $\pi_{\mathsf{pack-input}}$: The parties proceed as follows:

1. For each $i \in [M]$, the parties invoke f_{input} on x_i to obtain regular shares $[x_i]$.
2. The parties locally compute $\{\mathsf{block}_{0,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(0, \mathsf{layer}_0)$ and $\{\mathsf{block}_{1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(1, \mathsf{layer}_1)$.

3. For each $j \in [\sigma]$, the parties compute $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{1,j}, \mathsf{block}_{0,j})$.
4. For each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$, they compute packed secret sharing of vectors $\mathbf{x}^{j,q}$ and $\mathbf{y}^{j,q}$ as follows:

$$[\mathbf{x}^{j,q}] = \mathbb{F}_{\mathsf{SS-to-PSS}}(([x_{\mathsf{LeftInputs}_j[i]}])_{i \in \{(q-1)\ell+1,\ldots,q\ell\}})$$

$$[\mathbf{y}^{j,q}] = \mathbb{F}_{\mathsf{SS-to-PSS}}([y_{\mathsf{RightInputs}_j[i]}])_{i \in \{(q-1)\ell+1,\ldots,q\ell\}}$$

Output: Each party outputs its shares in $\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j \in [\sigma], q \in [w_{1,j}/\ell]}$.

Lemma 4. *This protocol securely computes $f_{\mathsf{pack-input}}$ with abort in the f_{input}-hybrid model in the presence of malicious adversaries who control at most t parties.*

Proof. The proof of this lemma follows trivially from the correctness of the non-interative transformation from regular secret sharing to packed secret sharing (Lemma 1).

C A Protocol for Generating Correlated Random Packed Sharings

In this section, we describe the protocol $\pi_{\mathsf{corr-rand}}$. The protocol proceeds as follows:
Auxiliary Inputs Vandermonde matrix $\mathbf{V}_{n,(n-t)} \in \mathbb{F}^{n \times (n-t)}$.
Inputs: All parties get a configuration block pair $(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$ as input.
Protocol $\pi_{\mathsf{corr-rand}}$: The parties proceed as follows:

— Each party P_i (for $i \in [n]$) chooses $3w_{m,j}/\ell$ random vectors $(\{\mathbf{s}_i^{q,\mathsf{mult}}, \mathbf{s}_i^{q,\mathsf{add}}, \mathbf{s}_i^{q,\mathsf{relay}}\}_{q \in [3w_{m,j}/\ell]}) \in \mathbb{F}^{\ell \times w_{m,j}/\ell}$ of length ℓ each.

— The parties compute $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.

— For each $q \in [w_{m+1,j}/\ell]$ and for each $k \in [\ell]$, let $e_{\mathsf{left}} = \mathsf{LeftInputs}_j[(q-1)\ell+i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}_j[(q-1)\ell+i]$ and the parties set:

$$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell \rfloor]$$

$$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell \rfloor]$$

where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k in block m, j is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$.

— For each $q \in [w_{m,j}/\ell]$ and $\mathsf{GateType} \in \{\mathsf{add}, \mathsf{relay}, \mathsf{mult}\}$, the parties compute

$$\langle \mathbf{s}_i^{q,\mathsf{GateType}} \rangle = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{GateType}}, n-1)$$

and for each $q \in [w_{m+1,j}/\ell]$, the parties compute

$$[\mathbf{s}_i^{q,\mathsf{left}}] = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{left}}, D), \quad [\mathbf{s}_i^{q,\mathsf{right}}] = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{right}}, D)$$

and sends the respective shares to each party.

— Given these shares, for each $q \in [w_{m,j}/\ell]$ and $\mathsf{GateType} \in \{\mathsf{add}, \mathsf{relay}, \mathsf{mult}\}$, the parties compute the following:

$$(\langle \mathbf{r}_1^{q,\mathsf{GateType}} \rangle, \ldots, \langle \mathbf{r}_{n-t}^{q,\mathsf{GateType}} \rangle) = \mathbf{V}_{n,(n-t)} \cdot (\langle \mathbf{s}_1^{q,\mathsf{GateType}} \rangle, \ldots, \langle \mathbf{s}_n^{q,\mathsf{GateType}} \rangle)$$

and for each $q \in [w_{m+1,j}/\ell]$, they compute

$$([\mathbf{r}_1^{q,\mathsf{left}}], \ldots, [\mathbf{r}_{n-t}^{q,\mathsf{left}}]) = \mathbf{V}_{n,(n-t)} \cdot ([\mathbf{s}_1^{q,\mathsf{left}}], \ldots, [\mathbf{s}_n^{q,\mathsf{left}}])$$

$$([\mathbf{r}_1^{q,\mathsf{right}}], \ldots, [\mathbf{r}_{n-t}^{q,\mathsf{right}}]) = \mathbf{V}_{n,(n-t)} \cdot ([\mathbf{s}_1^{q,\mathsf{right}}], \ldots, [\mathbf{s}_n^{q,\mathsf{right}}])$$

— The parties output their shares in $\{\{[\mathbf{r}_i^{q,\mathsf{left}}], [\mathbf{r}_i^{q,\mathsf{right}}]\}_{q \in [w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_i^{q,\mathsf{mult}} \rangle, \langle \mathbf{r}_i^{q,\mathsf{add}} \rangle, \langle \mathbf{r}_i^{q,\mathsf{relay}} \rangle\}_{q \in [w_{m,j}/\ell]}\}_{i \in [n-t]}$.

D Proof of our Semi-Honest Protocol

Lemma 5. *The protocol in Section 7 is private against a semi-honest adversary that corrupts upto t parties.*

Proof. Let $\mathcal{A}$ be the real adversary. We slightly abuse notation and let $\mathcal{A}$ also denote the set of corrupt parties. Let $\mathcal{H}$ denote the set of honest parties. We construct the simulator Sim as follows. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let Unique be as defined in Figure 6. Given the output of the protocol and inputs of the honest parties, for each $j \in [\sigma]$, the simulator proceeds as follows:

— **Input Sharing Phase:** It receives the shares $\{[\mathbf{x}^{j,q}]_{\mathcal{A}}, [\mathbf{y}^{j,q}]_{\mathcal{A}}\}_{j\in[\sigma],q\in[w_{q,j}/\ell]}$ that the adversary sends to $f_{\mathsf{pack-input}}$.

— **Circuit Evaluation:**

- *Correlated Randomness Generation:* For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$:
  - ∗ For each $i \in \mathcal{H}$, the simulator sends the following random shares

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q\in[w_{a+1}/\ell]}, \{\langle\mathbf{s}_i^{q,\mathsf{mult}}\rangle_{\mathcal{A}}, \langle\mathbf{s}_i^{q,\mathsf{add}}\rangle_{\mathcal{A}}, \langle\mathbf{s}_i^{q,\mathsf{relay}}\rangle_{\mathcal{A}}\}_{q\in[w_a/\ell]}$$

    to the adversary and receives the following shares from the adversary, for each $i \in \mathcal{A}$.

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{H}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{H}}\}_{q\in[w_{a+1}/\ell]}, \{\langle\mathbf{s}_i^{q,\mathsf{mult}}\rangle_{\mathcal{H}}, \langle\mathbf{s}_i^{q,\mathsf{add}}\rangle_{\mathcal{H}}, \langle\mathbf{s}_i^{q,\mathsf{relay}}\rangle_{\mathcal{H}}\}_{q\in[w_a/\ell]}.$$

  - ∗ For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares $\{\{[\mathbf{r}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q\in[w_{a+1}/\ell]}, \{\langle\mathbf{r}_i^{q,\mathsf{mult}}\rangle_{\mathcal{A}}, \langle\mathbf{r}_i^{q,\mathsf{add}}\rangle_{\mathcal{A}}, \langle\mathbf{r}_i^{q,\mathsf{relay}}\rangle_{\mathcal{A}}\}_{q\in[w_a/\ell]}\}_{i\in[n-t]}$, where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively. It then assigns these shares to different blocks in the circuit based on the configuration of each block. At the end of this step for each $m \in [d], j \in [\sigma]$, the simulator has the following shares:

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle\mathbf{r}_m^{j,q,\mathsf{mult}}\rangle_{\mathcal{A}}, \langle\mathbf{r}_m^{j,q,\mathsf{add}}\rangle_{\mathcal{A}}, \langle\mathbf{r}_m^{j,q,\mathsf{relay}}\rangle_{\mathcal{A}}\}_{q\in[w_{m,j}/\ell]}.$$

- *Layer-wise Circuit Evaluation:* For each $m \in [d], j \in [\sigma]$:
  - ∗ If $P_{\mathsf{leader}} \in \mathcal{H}$, the simulator simulates sending random shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to the adversary. It also uses $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to compute shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.
  - ∗ Else if $P_{\mathsf{leader}} \in \mathcal{A}$, the simulator simulates sending random shares $\{\langle\mathbf{z}_m^{j,q,\mathsf{mult}}\rangle_{\mathcal{H}}, \langle\mathbf{z}_m^{j,q,\mathsf{add}}\rangle_{\mathcal{H}}, \langle\mathbf{z}_m^{j,q,\mathsf{relay}}\rangle_{\mathcal{H}}\}_{q\in[w_{m,j}/\ell]}$ on behalf of the honest parties to the adversary. Based on the shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q\in[w_{m+1,j}/\ell]}$ sent by the adversary and $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$, the simulator computes $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.

— **Output Reconstruction:** For each $j \in [\sigma]$, using the output and previously computed shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{d+1,j}/\ell]}$, the simulator computes consistent shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q\in[w_{d+1,j}/\ell]}$ and sends them to the adversary.

View of the adversary generated by the simulator in the correlated randomness generation step is identically distributed to that in the real protocol. Moreover, as shown in [DN07], from super-invertibility property of the Vandermonde matrix, it follows that the $\mathbf{r}_i$ vectors generated by the parties at the end of the this step are random values that are unknown to any individual party or the adversary. Indistinguishability between the view of an adversary in the real protocol and the transcript generated by the simulator in the circuit evaluation step now follows from the privacy of packed secret sharing and the from the fact that the shares sent to the leader are of values that are masked by random values unknown to any party and hence appear completely random to the adversary.

E Additional Protocols for Malicious Security Compiler

In this section, we describe a few more sub-protocols that are needed for our maliciously secure protocol.

E.1 A Protocol for Generating Random Packed Shares

In this section, we describe the protocol $\pi_{\mathsf{pack-rand}}$ that securely realizes the functionality $f_{\mathsf{pack-rand}}$ (Figure 7). The protocol proceeds as follows:
Auxiliary Inputs Hyper-invertible matrix $\mathbf{H}_{n,n}$
Inputs: The parties do not have any inputs.
Protocol $\pi_{\mathsf{pack-rand}}$**:** The parties proceed as follows:

— If the parties wish to realize the independent mode of $f_{\mathsf{pack-rand}}$, each party P_i (for $i \in [n]$) chooses a random vector $\mathbf{u}_i \in \mathbb{F}^\ell$. It runs $\mathsf{pshare}(\mathbf{u}_i, D)$ to receive shares $[\mathbf{u}_i]$. For each $j \in [n]$, it party P_j, its share in $[\mathbf{u}_j]$.
— If the parties wish to realize the uniform mode of $f_{\mathsf{pack-rand}}$, each party P_i (for $i \in [n]$) chooses a random element $u_i \in \mathbb{F}$ and computes the vector $\mathbf{u}_i \in \mathbb{F}^\ell$ such that each element is u_i. It runs $\mathsf{pshare}(U_i, T)$ to receive shares $[\mathbf{u}_i]$. For each $j \in [n]$, it party P_j, its share in $[\mathbf{u}_j]$.
— Given shares $([\mathbf{u}_1], \ldots [\mathbf{u}_n])$, the parties compute

$$([\mathbf{r}_1], \ldots, [\mathbf{r}_n]) = \mathbf{H}_{n,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])$$

— Each party broadcasts its share in $[\mathbf{r}_{n-t+1}], \ldots, [\mathbf{r}_n]$ to the first $t+1$ parties. Those parties locally runs $\mathsf{open}([\mathbf{r}_{n-t+1}]), \ldots, \mathsf{open}([\mathbf{r}_n])$ to check if all the shares lie on the same degree D polynomial.
— If the parties wish to realize the uniform mode of $f_{\mathsf{pack-rand}}$, they additionally check that all $p(e_i)_{i\in[\ell]}$ are the same If these checks succeed, then the parties send "pass" to all other parties, else they send "fail".
— If each of the first $t+1$ parties output "pass", then the parties output their shares in $[\mathbf{r}_1], \ldots, [\mathbf{r}_{n-t}]$.

Output: The parties output $[\mathbf{r}_1], \ldots [\mathbf{r}_{n-t}]$.

Lemma 6. *This protocol securely computes $n-t$ instantiations of $f_{\mathsf{pack-rand}}$ with abort in the presence of malicious adversaries who controls t parties.*

The proof of this lemma follows from [DN07].

E.2 A Protocol for Checking Equality to Zero

In this section, we describe the protocol $\pi_{\mathsf{checkZero}}$ that securely realizes the functionality $f_{\mathsf{checkZero}}$ (Figure 8). The protocol proceeds as follows:
Inputs: The parties $\{P_i\}_{i\in[n]}$ hold shares $[v]$.
Protocol $\pi_{\mathsf{checkZero}}$**:** The parties proceed as follows:

— The parties $\{P_1, \ldots, P_n\}$ invoke f_{rand} to obtain sharings $[\mathbf{r}]$.
— The parties $\{P_1, \ldots, P_n\}$ invoke f_{mult} on $[\mathbf{r}]$ and $[\mathbf{v}]$ to obtain $[\mathbf{t}] = [\mathbf{r} \cdot \mathbf{v}]$.
— Each party P_i (for $i \in [n]$) send $\mathbf{t}_i$ to all other parties.
— Each party locally runs $\mathsf{open}([\mathbf{t}])$ on the revealed shares and checks if $\mathbf{t} = 0^\ell$. If so it outputs accept, else, it outputs reject.

We note that the proof of security for this protocol follows similarly to [CGH+18] and hence we omit it here.

E.3 A Protocol for Secure Packed Multiplication up to Linear Attacks

In this section, we describe the protocol $\pi_{\mathsf{pack-mult}}$ that securely realizes the functionality $f_{\mathsf{pack-mult}}$ (Figure 9). The protocol proceeds as follows:
Inputs: The parties $\{P_i\}_{i\in[n]}$ hold shares $[\mathbf{x}], [\mathbf{y}]$.
Protocol $\pi_{\mathsf{pack-mult}}$: The parties proceed as follows:

— The parties $\{P_1, \ldots, P_n\}$ locally compute $\langle \mathbf{x} \cdot \mathbf{y} \rangle = [\mathbf{x}] \cdot [\mathbf{y}]$
— the parties invoke $f_{\mathsf{pack-rand}}$ in independent mode to obtain packed secret shares $[\mathbf{r}]$ and $\langle \mathbf{r} \rangle$ for a random, independent vector $\mathbf{r}$.
— The parties the locally compute $\langle \mathbf{x} \cdot \mathbf{y} \rangle - \langle \mathbf{r} \rangle$ and send the resulting shares to the designated party P_{leader}.
— Party P_{leader} reconstructs all the values $\mathbf{x} \cdot \mathbf{y} - \mathbf{r}$. Party P_{leader} then generates a degree D sharing of this vector $[\mathbf{x} \cdot \mathbf{y} - \mathbf{r}]$ and send the resulting shares to all players
— Players locally compute $[\mathbf{z}] = [\mathbf{x} \cdot \mathbf{y} - \mathbf{r}] + [\mathbf{r}]$

Output: The parties output $[x \cdot y]$

Lemma 7. *This protocol securely computes $f_{\mathsf{pack-mult}}$ up to linear attacks in the $f_{\mathsf{pack-rand}}$-hybrid model, in the presence of malicious adversaries who controls t parties.*

Since this protocol is identical to the multiplication protocol of [DIK10], the proof of this lemma follows from the security proof given in [GIP15].

E.4 Protocol for Secure Dual Evaluation upto Linear Attacks

Lemma 8. *The protocol in Section 8.3 securely evaluates the circuit C on inputs $\mathbf{x}, \mathbf{rx}$ up to linear attacks in the presence of a malicious adversary who controls up to t parties.*

Proof (Sketch). [GIP15] show that any packed secret sharing based semi-honest protocol that satisfies the following properties, we know that any packed secret sharing based semi-honest protocol that satisfies the following three properties is secure against an adversary upto linear attacks:

— *T-randomization*: The messages sent by the honest parties to the corrupt parties (except in the last round), only depend on the randomness of the parties and not on their actual inputs.
— *Structure of the Last Round*: During the last round, only one party computes the output vector $\mathbf{z}$, as follows: let $F_{\mathcal{H}}$ and $F_{\mathcal{A}}$ be two linear functions, such that $z = F_{\mathcal{H}}(\mathsf{lmsg}_{\mathcal{H}}) + F_{\mathcal{A}}(\mathsf{lmsg}_{\mathcal{A}})$, where $\mathsf{lmsg}_{\mathcal{H}}$ are the messages sent by the honest parties in the last round and $\mathsf{lmsg}_{\mathcal{A}}$ are the messages sent by the corrupt parties in the last round.
— *Privacy of the last round*: The distribution of the messages $\mathsf{lmsg}_{\mathcal{H}}$ sent by the honest parties in the last round are uniform, conditioned on $F_{\mathcal{H}}(\mathsf{lmsg}_{\mathcal{H}}) = \mathbf{z} - F_{\mathcal{A}}(\mathsf{lmsg}_{\mathcal{A}})$.

The first property is trivially satisfied by our sub-protocol — indeed, Genkin's thesis [Gen16] already shows that semi-honest [DIK10] satisfies the first property, and those arguments generalize to our sub-protocol in a straightforward way. The second property is also easy to verify, indeed the masked output vector is computed by P_{leader} in our subprotocol, by running the reconstruction algorithm of packed secret sharing, which is a linear function. The third property is also satisfied by our protocol, since the shares sent by the parties in the last round to P_{leader} correspond to shares for a degree $n-1$ polynomial, the shares of the honest parties are uniformly distributed given the output and the shares of the corrupt parties. As a result, our protocol securely evaluates C on inputs $\mathbf{x}, \mathbf{rx}$ up to linear attacks.

F Security Proof for our Maliciously Secure Protocol

Lemma 9. *If $\mathcal{A}$ sends a non-zero linear attack value in any of the calls to f_{mult} or $f_{\mathsf{dual-eval}}$ in the execution of the protocol given in Section 8.5, then the vector $\mathbf{t}$ in the verification stage equals a 0-vector with probability less that $2/|\mathbb{F}|$.*

Proof. A malicious adversary can carry out linear attacks on f_{mult} and $\pi_{\mathsf{dual-eval}}$, meaning that the adversary can add an arbitrary linear combination of the input wires of a gate to the value on its outgoing wire. We show that the technique used by Chida et al., for detecting addditive errors can be used in the packed secret sharing setting to detect linear attacks. Since we perform a check on packed shares as opposed to regular shares, our check can be viewed as ℓ parallel checks at the end. We essentially end up computing ℓ different linear combinations of approximately $2|C|/\ell$ values. Given our description of $f_{\mathsf{checkZero}}$, it is clear that if any of these checks fail, $f_{\mathsf{checkZero}}$ will output $\perp$. Therefore, for exact probability calculation, we bound the probability of the adversary injecting errors and getting away in any one of the linear combinations. More specifically, we consider the linear combination over the first elements in each packed sharing output. For each $m \in [d+1]$ in the circuit, our protocol generates $4w_m/\ell$ packed shares of the form $\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}], [\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{rz}_m^{j,q,\mathsf{right}}]\}_{j\in[\sigma],q\in[w_{d,j}/\ell]}$. We simplify the notation and let the set of packed shares on each layer m, be of the form $\{[\mathbf{z}_m^q], [\mathbf{rz}_m^q]\}_{q\in[w_m/\ell]}$. Similarly, we use $\boldsymbol{\alpha}_m^q$ to denote the $\boldsymbol{\alpha}$ vectors corresponding to these packed secret sharings.

Finally, we slightly abuse notation and let z_m^q denote the first element in the vector $\mathbf{z}_m^q$ and α_m^q to denote the first element in the vector $\boldsymbol{\alpha}_m^q$.

We use different variables to denote the additive errors that the adversary can inject on each of these computations.

— Let $\mathbf{F}_m^q : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be the linear error function induced as a result of operating on incorrectly computed shares of $\mathbf{z}_m^q$. For instance, when the vectors $\mathbf{z}_m^q$ and $\boldsymbol{\alpha}_m^q$ are multiplied, a linear error of the form $\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)$ is induced on the first element of the output vector. We note that since $\boldsymbol{\alpha}_m^q$ in our protocol is guaranteed to be honestly secret shared, no error of the form $\mathbf{L}(\mathbf{z}_m^q)$ is induced when multiplying $\boldsymbol{\alpha}_m^q$ and $\mathbf{z}_m^q$.

— Similarly, let $\mathbf{G}_m^q : \mathbb{F}^\ell \rightarrow \mathbb{F}$ be the linear error function induced as a result of operating on incorrectly computed shares of $\mathbf{rz}_m^q$.

— We let f_m^q be the resultant linear error on z_m^q and g_m^q be the resultant linear error on rz_m^q. We note that these errors are not arbitrary values but a linear combination of the input values to the gates in layer m.

Recall that if every party behaves honestly, then

$$u = \sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_m^q z_m^q \text{ and } v = \sum_{m\in[d+1]} \sum_{m\in[w_m/\ell]} \alpha_q(rz_m^q)$$

We would like to check if $ru = v$, ie.

$$r \sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_m^q z_m^q = \sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_q(rz_m^q)$$

This is trivially true, if no errors were introduced by the adversary at any step. Accounting for all the linear errors that the adversary might introduce, we get

$$ru = r\left(\sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_m^q(z_m^q + f_m^q) + \mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)\right)$$

$$v = \left(\sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_m^q(rz_m^q + g_m^q) + \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q)\right)$$

We want to calculate the probability that the following equation holds, i.e.,

$$r\left[\left(\sum_{m\in[d+1]}\sum_{q\in[w_m/\ell]}\alpha_m^q(z_m^q+f_m^q)+\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)\right)\right]=\left(\sum_{m\in[d+1]}\sum_{q\in[w_m/\ell]}\alpha_m^q(rz_m^q+g_m^q)+\mathbf{G}_m^q(\boldsymbol{\alpha}_m^q)\right)$$

In other words,

$$\sum_{m\in[d+1]}\sum_{q\in[w_m/\ell]}\alpha_m^q(rf_m^q-g_m^q)=\sum_{m\in[d+1]}\sum_{q\in[w_m/\ell]}\mathbf{G}_m^q(\boldsymbol{\alpha}_m^q)-r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)$$

We now consider the following cases:

— **Case 1:** All the inputs, intermediate wire computations and $\alpha's$ were honestly secret shared and computed. And the errors were only introduced during the verification step. Since the verification step in this case corresponds to multiplying honestly secret shared vectors, the only kind of errors that the adversary can introduce in the case are arbitrary additive errors, that are not correlated to any of the input values. Let d_u be the cumulative additive error on the computation of u, ans d_v be the cumulative additive error on the computaiton of v. We can re-write the above equation as $0=\sum_{m\in[d+1]}\sum_{q\in[w_m/\ell]}\mathbf{G}_m^q(\boldsymbol{\alpha}_m^q)-r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)=d_u-rd_v$.
Since r is sampled uniformly, the probability that $d_u-rd_v=0$ is $1/|\mathbb{F}|$, if either $d_u\neq 0$ or $d_v\neq 0$.

— **Case 2:** $\exists m\in[d],\exists q\in[w_m/\ell]$, such that the output $\mathbf{z}_m^q$ was not correctly secret shared. Let m_0 be the smallest such m and q_0 be the smallest such q. We want to calculate the probability that the following equation holds, i.e.,

$$\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0})-r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q)=\alpha_{m_0}^{q_0}(rf_{m_0}^{q_0}-g_{m_0}^{q_0})+\sum_{m\in[d+1],m\neq m_0}\sum_{q\in[w_m/\ell]}r\alpha_m^q(f_m^q-g_m^q)-\mathbf{G}_m^q(\boldsymbol{\alpha}_m^q)+r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)$$

- If $\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0})-r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q)\neq 0$: Since all the $\alpha's$ (including $\alpha_{m_0}^{q_0}$) are generated honestly and are unknown to the adversary, the above equality holds only with probability $1/|\mathbb{F}|$.
- If $\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0})-r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q)=0$: Since r is sampled uniformly at random, this only happens with probability $1/|\mathbb{F}|$.

Hence, overall the probability that that the view generated by the simulator in Case 2 is distinguishable from the view in the real execution is at most

$$\frac{1}{|\mathbb{F}|}+\left(1-\frac{1}{|\mathbb{F}|}\right)\frac{1}{|\mathbb{F}|}<\frac{2}{|\mathbb{F}|}$$

In both cases, the probability of distinguishability is upper bounded by $\frac{2}{|\mathbb{F}|}$.

Operating over Smaller fields. This protocol works for fields that are large enough such that $\frac{2}{|\mathbb{F}|}$ is an acceptable probability of an adversary cheating. In cases where it might be desirable to instead work in a smaller field, we can use the same approach as used by Chida et al. [CGH+18]. In particular, instead of having a single randomized evaluation of the circuit w.r.t. r, we can generate shares for δ random values $r_1,\ldots,r_\delta$ (such that $(\frac{2}{|\mathbb{F}|})^\delta$ is negligible) and run multiple randomized evaluations of the circuit and verification steps for each r_i. Since each r is independently sampled and their corresponding verification procedures are also independent, this will yield a cheating probability of at most $(\frac{2}{|\mathbb{F}|})^\delta$, as required.

Given this lemma, we now prove the following Theorem.

Theorem 1. *Let k be a statistical security parameter, and let $\mathbb{F}$ be a finite field such that $(3/|\mathbb{F}|)^\delta\leq 2^{-k}$, for some $\delta\geq 1$. Let f be an n-party functionality over $\mathbb{F}$. Then, there exists a protocol in the $(f_{\mathsf{pack-input}},f_{\mathsf{pack-rand}},f_{\mathsf{dual-eval}},f_{\mathsf{mult}},f_{\mathsf{checkZero}})$-hybrid model with statistical error 2^{-k}, in the presence of a malicious adversary controlling t parties.*

Proof. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let $\mathcal{A}$ be an adversary in the real world. As before we use $\mathcal{A}$ to also denote the set of corrupted parties. The simulator Sim in the ideal world initializes a variable $\mathsf{flag} = 0$ and proceeds as follows:

1. **Input Sharing:** Sim receives from $\mathcal{A}$ the set of corrupted inputs and the shares of corrupted parties $\{[\mathbf{z}^{j,q,\mathsf{left}}]_\mathcal{A}, [\mathbf{z}^{j,q,\mathsf{right}}]_\mathcal{A}\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ that the adversary sends to the $f_{\mathsf{pack-input}}$ functionality. It reconstructs these inputs and saves them.
2. **Preprocessing:** The simulator receives the shares $[\mathbf{r}]_\mathcal{A}$ and $\{[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}]_\mathcal{A}, [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}]_\mathcal{A}\}_{m\in[d],j\in[\sigma],q\in[w_{m,j}/\ell]}$ of the corrupted parties that the adversary sends to $f_{\mathsf{pack-rand}}$.
3. **Randomization of inputs:** For each $j \in [\sigma], q \in [2w_{1,j}/\ell]$, the simulator Sim plays the role of f_{mult} in the multiplication of vectors $\mathbf{z}^{j,q,\mathsf{left}}$ and $\mathbf{z}^{j,q,\mathsf{right}}$ with $[\mathbf{r}]$. Specifically, Sim hands the corrupted parties shares in $[\mathbf{r}]$, $\mathbf{z}^{j,q,\mathsf{left}}$ and $\mathbf{z}^{j,q,\mathsf{left}}$ to the adversary. Upon receiving the linear error function and the corrupted parties shares of the resulting vector, the simulator stores all the corrupted parties' shares. If any linear error function was received, it sets $\mathsf{flag} = 1$.
4. **Dual Circuit Evaluation:** As discussed in the main protocol, dual circuit evaluation requires the parties to run π_{eval} twice – on actual inputs and on randomized inputs. Here we describe how the transcript for evaluation on actual inputs it simulated. The transcript on randomized inputs is simulated in almost exactly the same way, and hence we omit it here.
   - *Correlated Randomness Generation for circuit evaluation on actual inputs:* For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$:
     - For each $i \in \mathcal{H}$, the simulator sends the following random shares
     $$\{[\mathbf{s}_i^{q,\mathsf{left}}]_\mathcal{A}, [\mathbf{s}_i^{q,\mathsf{right}}]_\mathcal{A}\}_{q\in[w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}}\rangle_\mathcal{A}, \langle \mathbf{s}_i^{q,\mathsf{add}}\rangle_\mathcal{A}, \langle \mathbf{s}_i^{q,\mathsf{relay}}\rangle_\mathcal{A}\}_{q\in[w_a/\ell]}$$
     to the adversary and receives the following shares from the adversary, for each $i \in \mathcal{A}$.
     $$\{[\mathbf{s}_i^{q,\mathsf{left}}]_\mathcal{H}, [\mathbf{s}_i^{q,\mathsf{right}}]_\mathcal{H}\}_{q\in[w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}}\rangle_\mathcal{H}, \langle \mathbf{s}_i^{q,\mathsf{add}}\rangle_\mathcal{H}, \langle \mathbf{s}_i^{q,\mathsf{relay}}\rangle_\mathcal{H}\}_{q\in[w_a/\ell]}.$$
     - The simulator computes $\mathsf{LeftInputs}, \mathsf{RightInputs} = \mathsf{WireConfiguration}(\mathsf{block}_{a+1}, \mathsf{block}_a)$. For each $q \in [w_{a+1}/\ell]$ and for each $k \in [\ell]$, it sets $e_{\mathsf{left}} = \mathsf{LeftInputs}[(q-1)\ell+i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[(q-1)\ell+i]$. For each $i \in \mathcal{A}$, the simulator checks if
     $$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell\rfloor]$$
     $$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell\rfloor]$$
     where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k in block a is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$. If any of these checks fail for any $i \in \mathcal{A}$, the simulator sets $\mathsf{flag} = 1$.
     - For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares $\{\{[\mathbf{r}_i^{q,\mathsf{left}}]_\mathcal{A}, [\mathbf{r}_i^{q,\mathsf{right}}]_\mathcal{A}\}_{q\in[w_{a+1}/\ell]}, \{\langle \mathbf{r}_i^{q,\mathsf{mult}}\rangle_\mathcal{A}, \langle \mathbf{r}_i^{q,\mathsf{add}}\rangle_\mathcal{A}, \langle \mathbf{r}_i^{q,\mathsf{relay}}\rangle_\mathcal{A}\}_{q\in[w_a/\ell]}\}_{i\in[n-t]}$, where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively.
     - It then assigns these shares to different blocks in the circuit based on the configuration of each block. At the end of this step for each $m \in [d], j \in [\sigma]$, the simulator has the following shares:
     $$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_\mathcal{A}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_\mathcal{A}\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_m^{j,q,\mathsf{mult}}\rangle_\mathcal{A}, \langle \mathbf{r}_m^{j,q,\mathsf{add}}\rangle_\mathcal{A}, \langle \mathbf{r}_m^{j,q,\mathsf{relay}}\rangle_\mathcal{A}\}_{q\in[w_{m,j}/\ell]}.$$
   - *Circuit evaluation on actual inputs:* The simulator computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$. For each $q \in [w_{m+1,j}/\ell]$ and for each $k \in [\ell]$, it sets $e_{\mathsf{left}} = \mathsf{LeftInputs}[(q-1)\ell+i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[(q-1)\ell+i]$. For each $i \in \mathcal{A}$, the simulator check if
   $$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell\rfloor]$$
   $$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell\rfloor]$$
   where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k on layer m is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$. If any of these checks fail for any $i \in \mathcal{A}$, the simulator sets $\mathsf{flag} = 1$.

— For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_m^{j,q,\mathsf{mult}}\rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{add}}\rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{relay}}\rangle_{\mathcal{A}}\}_{q\in[w_{m,j}/\ell]}.$$

— If $P_{\mathsf{leader}} \in \mathcal{H}$:

(a) The simulator receives shares $\{\langle \mathbf{z}_m^{j,q,\mathsf{mult}}\rangle_{\mathcal{A}}, \langle \mathbf{z}_m^{j,q,\mathsf{add}}\rangle_{\mathcal{A}}, \langle \mathbf{z}_m^{j,q,\mathsf{relay}}\rangle_{\mathcal{A}}\}_{q\in[w_{m,j}/\ell]}$ from the adversary.

(b) For each $i \in \mathcal{A}$, the simulator checks if $\langle \mathbf{z}_m^{j,q,\mathsf{mult}}\rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i \cdot [\mathbf{z}_m^{j,q,\mathsf{right}}]_i + \langle \mathbf{r}_m^{j,q,\mathsf{mult}}\rangle_i$ and $\langle \mathbf{z}_m^{j,q,\mathsf{add}}\rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i + [\mathbf{y}_m^{j,q,\mathsf{right}}]_i + \langle \mathbf{r}^{j,q,\mathsf{add}}\rangle_i$ and $\langle \mathbf{z}_m^{j,q,\mathsf{relay}}\rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i + \langle \mathbf{r}_m^{j,q,\mathsf{relay}}\rangle_i$. If any of these checks fail, the simulator sets $\mathsf{flag} = 1$.

(c) The simulator simulates sending random shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to the adversary.

(d) The simulator uses $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to compute shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.

— Else if $P_{\mathsf{leader}} \in \mathcal{A}$:

(a) The simulator simulates sending random shares $\{\langle \mathbf{z}_m^{j,q,\mathsf{mult}}\rangle_{\mathcal{H}}, \langle \mathbf{z}_m^{j,q,\mathsf{add}}\rangle_{\mathcal{H}}, \langle \mathbf{z}_m^{j,q,\mathsf{relay}}\rangle_{\mathcal{H}}\}_{q\in[w_{m,j}/\ell]}$ on behalf of the honest parties to the adversary.

(b) The simulator uses the shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q\in[w_{m+1,j}/\ell]}$ sent by the adversary to reconstruct $\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}$ and $\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}$ and checks if these are consistent with the previously sent and computed shares. If not, it sets $\mathsf{flag} = 1$.

(c) Finally, the simulator uses $\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}$, $\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}$ and $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$, to compute $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.

5. **Verification Step:** The simulator simulates the honest parties sending their shares in the opening of $[\mathbf{r}]$ to the adversary and receives the shares that the adversary sends to the honest parties in this open. If any honest party would abort, then the simulator simulates it by sending $\perp$ to all parties, and to the trusted functionality and halts. Finally, Sim simulates $f_{\mathsf{checkZero}}$ as follows, If $\mathsf{flag} = 1$, then Sim simulates $f_{\mathsf{checkZero}}$ by sending reject and then all honest parties sending $\perp$. Otherwise, Sim proceeds to the next step.
6. **Output Reconstruction:** If no abort had occurred, Sim sends the inputs of the adversary that it had extracted from the input sharing phase to the ideal functionality computing f. It receives back the output from the ideal functionality. The simulator then computes the shares of the honest parties using this output and the shares of the corrupt parties (that it can compute based on the information it has). It sends these shares to the adversary as part of the output reconstruction phase.
It then receives messages from the adversary. It uses these messages to reconstruct the output for the honest party. If for any honest party, this reconstruction fails, it sends $\perp$ along with the identity of the honest party to the ideal functionality, signaling it to send $\perp$ to that party.

Overall, the view of the adversary in the ideal world is identical to its view in the real world, except with $3/|\mathbb{F}|$ probability. Up until the correlated randomness generation step of the dual circuit evaluation phase, the view of the adversary generated by the simulator is identically distributed to that in the real world. From super-invertibility property [DN07] of Vandermonde matrices, it follows that the $\mathbf{r}_i$ vectors generated by the parties at the end of this step are random and unknown to the adversary. However, since the adversary is malicious, these vectors may not have the right correlation. Nevertheless, indistinguishability between the view of an adversary in the real protocol and the transcript generated by the simulator up until the verification step now follows from the privacy of packed secret sharing and the from the fact that the shares sent to the leader are of values that are masked by random values unknown to any party and hence appear completely random to the adversary.

In case no errors are introduced in the protocol before and during the verification step, then the only difference between the real and ideal executions is that the input shares of the honest parties are set to

0 in the simulated transcript. However, by the perfect secrecy of packed secret sharing, this has the same distribution as in a real execution.

In case, some errors were introduced, then the simulator always simulates $f_{\mathsf{checkZero}}$ outputting reject. However, in the real execution, the probability that vector sent to $f_{\mathsf{checkZero}}$ is a non-zero vector is at most $2/|\mathbb{F}|$ and if indeed a non-zero vector is sent to $f_{\mathsf{checkZero}}$, it will get detected except with probability $1/|\mathbb{F}|$. Thus overall, in this case the adversary can avoid detection with probability at most $3/|\mathbb{F}|$. Since this is the only difference between the real execution and the ideal simulation, we have that the statistical difference between these distributions is less than $3/|\mathbb{F}|$.