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, Baltimore, USA,
{gbeck,aarushig,abhishek}@cs.jhu.edu
[2] Boston University, Boston, USA, 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) [39,23,6,4] 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 [27,12,30,7,35,16].

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.

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

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 [34] and TOR [36,13] 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 [38], enabling novel measurements.

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 [28], Ishai et al. showed that zero-knowledge (ZK) proofs [24] 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:

[4] SIMD circuits are arithmetic circuits that simultaneously evaluate ℓ copies of the same arithmetic circuit on different inputs. Genkin et al. [20] showed that it is possible to design an $O(|\mathsf{C}|)$ MPC protocol for SIMD circuits, where $\ell = \Theta(n)$.

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 [4,6].

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 [12]. 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) [15], 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 [9] and [12].

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 [11] and [10] 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 [20].)

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 [11,10,20]; 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 [7,35,16].

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 [21,20,30,7,35,16,25] has been done on designing efficient malicious security compilers for honest majority MPC. With the exception of [20], all of these works design compilers for protocols based on regular secret sharing (SS) as opposed to PSS. The most recent of these works [7,35,16,25] 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 [7,35,16,25] are not directly applicable to our protocols. Nevertheless, borrowing upon the insights from [20], we demonstrate that the techniques developed in [7] 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 1.7–3, 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 6.2 and 6.3.

[5] We note that for more commonly used corruption thresholds $n/2 > t > n/4$, the overhead incurred by our compiler is somewhere between 2.5–3.

For $n = 125$ parties and $t < n/3$, our malicious secure protocol only requires each party to, on average, communicate approximately $3\frac{1}{4}$ *field elements per gate* of a highly repetitive circuit. In contrast, the state-of-the-art [16] (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 [16] for circuits with around 75% 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. [28] 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 [3] via the Fiat-Shamir paradigm [14]. Recent works have demonstrated the practical viability of this approach by constructing zero-knowledge proofs [22,5,29,2] 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 [29]. Importantly, however, it can have more efficient prover and verifier computation time. This is because [29] 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 [2] 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.

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.

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 [7], incurs a multiplicative overhead of somewhere between 2.5 and 3, over the semi-honest protocol. Recent works of [16,25] achieve even lower overheads than the compiler of [7]. Another useful direction would be to integrate our ideas with the techniques in [16,25] (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 [4], leverage Shamir's secret sharing [37] 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 [12], 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 [15], 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

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

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 $XY = (x_1 y_1, x_2 y_2, x_3 y_3)$ 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 $\Theta(n)$ 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 $x_1 y_2$ 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.
— 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, [10,20] 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

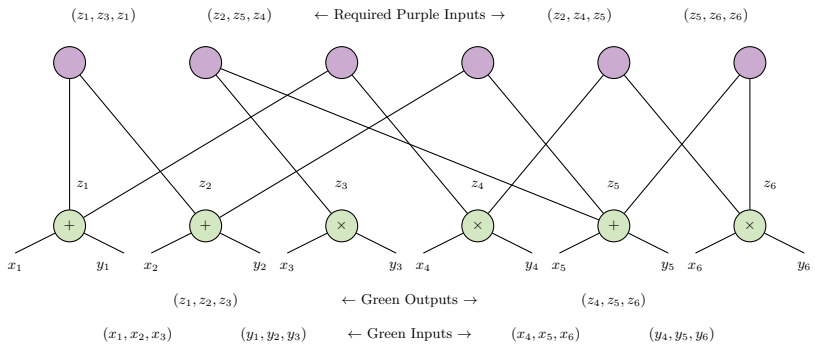

Fig. 1: A simple example pair of circuit layers illustrating the need for differing-operation packed secret sharing and our realignment procedure. Players begin by evaluating both addition and multiplication on each pair of input vectors. However, the resulting vectors are not properly aligned to compute 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 $(z_1^{add}, z_3^{mult}, z_1^{add}), (z_2^{add}, z_5^{add}, z_4^{mult}), (z_2^{\text{add}}, z_4^{mult}, z_5^{\text{add}})$, and $(z_5^{\text{add}}, z_6^{mult}, z_6^{mult})$

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 packed 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 $(x_1 y_1, x_2 y_2, x_3 y_3)$. 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 $x_1 + y_1, x_2 + y_2, x_3 + y_3, x_1 y_1, x_2 y_2, x_3 y_3$ (while masked). The leader then performs this "selection" process, and packs only the required values to get a vector $(x_1 y_1, x_2 + y_2, x_3 y_3)$, and discards the unused values $x_1 + y_1, x_2 y_2, x_3 + y_3$. 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.

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^{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 $(z_1^{add}, z_3^{mult}, z_1^{add})$ (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,

[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.

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 6.1.

However, 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 (see Section 5) 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 $O(n|\mathsf{C}|)$ 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 [10] 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 [10] is primarily concerned with asymptotic efficiency, such circuit modification strategies are sufficient for their work.

Comparison with [10]. We briefly recall the strategy used in [10], 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 [21]. 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 [7,35,16,25], introduce only a small multiplicative overhead (e.g., 2 in the case of [7]) and require only a constant number of additional rounds to perform a single, consolidated check

Genkin et al. showed in [20] (with additional technical details in [19]) 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 [7]. The resulting maliciously secure protocol has twice the complexity of our semi-honest protocol, 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 [7], 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 a circuit C, 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 and regular threshold secret sharing scheme, 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 $2D$ 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 5 for details).

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 $\mathsf{LeftInputs}[j] = l(g_j)$ and $\mathsf{RightInputs}[j] = r(g_j)$
3. Output $\mathsf{LeftInputs}, \mathsf{RightInputs}$.

Fig. 2: A function that computes a proper alignment for evaluating 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 $\{\mathsf{block}^{\mathsf{child}}_{m,j}\}$ 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 $\mathsf{block}_{m,j}$ consists 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 [11,10,20] 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 $\texttt{min}(O(A), O(B))$ 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 m^{th} layer and $w_{m,j}$ to denote the width of $\mathsf{block}_{m,j}$.

4.3 Examples of Highly Repetitive Circuits

We give brief overviews of three functionalities with circuit representations that are highly repetitive for up to a large number of parties. Extended discussion of these applications is included in the full version of this paper.

Machine Learning. Machine learning algorithms extract trends from large datasets to facilitate accurate prediction in new, unknown circumstances. A common family of algorithms for training machine learning models is "gradient descent." This algorithm iteratively reduces the models error by making small, greedy changes, terminating when the model quality plateaus. When run with MPC, the number of iterations must be data oblivious and cover the worst case scenario. For a more complete description of gradient descent training algorithms, and their adaptation to MPC, see [31].

It is difficult to compute the exact number of gates for privacy-preserving model training in prior work. In one of the few concrete estimates, Gascón et al. [18] realize coordinate gradient descent training algorithms with approximately 10^{11} gates, which would take 3000GB to store [32]. Subsequent work instead built a library of sub-circuits that could be loaded as needed. As the amount of data used to train models continues to grow, circuit 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; gradient decent algorithm is many iterations of the same functionality. In the implementation of Mohassel et al. [31], the default configuration for training is 10000 iterations, deep enough to accommodate massive numbers of players. Indeed, in the worst case the depth of a gradient descent algorithm must be linear in the input size. This is because gradient descent usually uses a *batching* technique, in which the input data is partitioned into batches and run through the algorithm one at a time.

The width of gradient descent training algorithms is usually roughly proportional to the dimension of the dataset, which is usually quite high for interesting applications. We note that if the width of the data is no wide enough, the natural parallelism of gradient decent training algorithms can be leveraged to provide

Table 1: Size of the highly repetitive circuits we consider in this work. We compile these functions into $\mathbb{F}_2$ circuits using Frigate [33] (containerized by [26]). 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	—	$\sim 100\%$

more width: it is typical to use a *random restart* strategy to avoid getting trapped at local minima, each of which can be execute in parallel.

Cryptographic Hash Functions. All currently deployed cryptographic hash functions rely on iterating over a round function, each iteration of which 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.

Consider SHA256 [1], one of the most widely deployed hash functions; given its common use in applications like Bitcoin [34] and ECDSA [17], 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. We compiled SHA256 for a single block of input into a circuit using Frigate [33]. 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 additional blocks of data must be hashed, the percentage of the circuit that is repeated structure will be higher. For example, if there are as few as 10 blocks of input, the circuit is already 97% repeated structure. Common applications of hash functions, like computing a Merkle tree over player inputs, run hash functions in parallel, ensuring there is sufficient width for accommodate large numbers of parties.

Block Ciphers. Modern block ciphers, similar to cryptographic functions, are iterative by nature. For example, Advanced Encryption Standard uses either 10, 12, or 14 iterations of its round function, depending on key length. 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.

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 $\Theta(n)$. 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 [7] and [16] 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. 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 leader can reconstruct and re-share normal shares as necessary. Importantly, since all these protocols are linear, it's still possible to use the malicious security compiler of [7].

5 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 non-interactive. 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 Shamir secret sharing (using degree $t + \ell$ polynomials), 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. We require each $p_i(z)$ (for $i \in [\ell]$) to be 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 $s_i \in \mathbb{F}$ 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 j^{th} 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 $p_i(z) = s_i + q_i(z)\prod_{j=1}^{\ell}(z-e_j)$, 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:

$$p(z)=\sum_{i=1}^{\ell} p'_i(z)L_i(z)+s_iL_i(z)=\sum_{i=1}^{\ell} p'_i(z)\prod_{j=1,j\neq i}^{\ell}\frac{(z-e_j)}{(e_i-e_j)}+\sum_{i=1}^{\ell} s_iL_i(z)$$
$$=\sum_{i=1}^{\ell} q_i(z)\prod_{j=1,j\neq i}^{\ell}\frac{(z-e_j)}{(e_i-e_j)}\prod_{j=1}^{\ell}(z-e_j)+\sum_{i=1}^{\ell} s_iL_i(z)$$

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

$$p(z)=\sum_{i=1}^{\ell} q'_i(z)\prod_{j=1}^{\ell}(z-e_j)+\sum_{i=1}^{\ell} s_iL_i(z)=q(z)\prod_{j=1}^{\ell}(z-e_j)+\sum_{i=1}^{\ell} s_iL_i(z)$$

where $q(z) = \sum_{i=1}^{\ell} q'_i(z)$ 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 $\mathbf{v} = (s_1, \ldots, s_\ell)$.

6 Our Order-C Protocols

6.1 Sub-Functionalities and Protocols

Both our semi-honest and maliciously secure protocols depend on a number of sub-functionalities and protocols which we present in this section.

$f_{\mathsf{pack-input}}$ **functionality.** This functionality takes in the inputs of the players and outputs packed secret shares. 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. A detailed description of this functionality appears in Figure 3. The description of a protocol that makes use of our non-interactive packing protocol from Section 5, that securely realizes this functionality is deferred to the full-version of this paper.

$f_{\mathsf{corr-rand}}$ **functionality.** This functionality generates correlated randomness for our main construction. Recall from the technical overview that the values in the packed secret shares of random values must be generated according to the circuit structure. More specifically, the unmasking values (degree

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. 3: Packed Secret sharing of all inputs functionality

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 functionality, we quickly note the number of shares generated, 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 in a packed secret share. Each of these operations must be masked with different randomness to ensure privacy. As such, the functionality generates $3w_{m,j}/\ell$ shares of uniformly random vectors. To facilitate unmasking after the leader has run the realignment procedure, the functionality must generate shares of vectors with values selected from these $3w_{m,j}/\ell$ uniformly random vectors. This selection is governed by $\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$. As 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, this is $(3w_{m,j} + 2w_{m+1,j})/\ell$ packed secret sharings. A detailed description of this functionality appears in Figure 4. The description of a protocol that securely realizes this functionality is deferred to the full-version of our paper.

π_{layer} **Protocol.** This sub-protocol takes properly aligned input vectors $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ 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 $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$ that can be used to evaluate the next layer. We note that for notational convenience, this 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,

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

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

- Each honest party sends $\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j}$ to the functionality.
- The ideal simulator Sim sends $\{\mathbf{u}_i^{q,\mathsf{left}}, \mathbf{u}_i^{q,\mathsf{right}}\}_{q\in[w_{m+1}/\ell]}$ and $\{\mathbf{v}_i^{q,\mathsf{mult}}, \mathbf{v}_i^{q,\mathsf{add}}, \mathbf{v}_i^{q,\mathsf{relay}}\}_{q\in[w_{m,j}/\ell]}$ for each corrupt party $i \in \mathcal{A}$.
- The functionality $f_{\mathsf{corr-rand}}$ samples random vectors $(\{\mathbf{r}^{q,\mathsf{mult}}, \mathbf{r}^{q,\mathsf{add}}, \mathbf{r}^{q,\mathsf{relay}}\}_{q\in[w_{m,j}/\ell]}) \in \mathbb{F}^{\ell\times 3w_{m,j}/\ell}$ of length ℓ and does the following:
  - For each $q \in [w_{m+1,j}/\ell]$, it sets $[\mathbf{r}^{q,\mathsf{left}}]_{\mathcal{A}} = \{\mathbf{u}_i^{q,\mathsf{left}}\}_{i\in\mathcal{A}}$ and $[\mathbf{r}^{q,\mathsf{right}}]_{\mathcal{A}} = \{\mathbf{u}_i^{q,\mathsf{right}}\}_{i\in\mathcal{A}}$. .
  - For each $q \in [w_{m,j}/\ell]$, it sets $\langle\mathbf{r}^{q,\mathsf{mult}}\rangle_{\mathcal{A}} = \{\mathbf{v}_i^{q,\mathsf{mult}}\}_{i\in\mathcal{A}}$ and $\langle\mathbf{r}^{q,\mathsf{add}}\rangle_{\mathcal{A}} = \{\mathbf{v}_i^{q,\mathsf{add}}\}_{i\in\mathcal{A}}$ and $\langle\mathbf{r}^{q,\mathsf{relay}}\rangle_{\mathcal{A}} = \{\mathbf{v}_i^{q,\mathsf{relay}}\}_{i\in\mathcal{A}}$.
  - It computes $\mathsf{LeftInputs}, \mathsf{RightInputs} = \mathsf{WireConfiguration}(\mathsf{block}_{m\to m+1})$.
  - For each $q \in [w_{m+1,j}]$ and for each $k \in [\ell]$, let $e_{\mathsf{left}} = \mathsf{LeftInputs}[(q-1)\ell + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[(q-1)\ell + i]$ and set $\mathbf{r}^{q,\mathsf{left}}[k] = \mathbf{r}^{\lfloor e_{\mathsf{left}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell\rfloor]$ and $\mathbf{r}^{q,\mathsf{right}}[k] = \mathbf{r}^{\lfloor e_{\mathsf{right}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell\rfloor]$, 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, $\mathsf{GateType}_k = \mathsf{relay}$.
  - For each $q \in [w_{m,j}/\ell]$, it runs $\mathsf{pshare}(\mathbf{r}^{q,\mathsf{mult}}, \mathcal{A}, \langle\mathbf{v}^{q,\mathsf{mult}}\rangle_{\mathcal{A}}, 2D)$, $\mathsf{pshare}(\mathbf{r}^{q,\mathsf{add}}, \mathcal{A}, \langle\mathbf{v}^{q,\mathsf{add}}\rangle_{\mathcal{A}}, 2D)$, $\mathsf{pshare}(\mathbf{r}^{q,\mathsf{relay}}, \mathcal{A}, \langle\mathbf{v}^{q,\mathsf{relay}}\rangle_{\mathcal{A}}, 2D)$.
  - For each $q \in [w_{m+1,j}/\ell]$, it runs $\mathsf{pshare}(\mathbf{r}^{q,\mathsf{left}}, \mathcal{A}, [\mathbf{u}^{q,\mathsf{left}}]_{\mathcal{A}}, D)$ and $\mathsf{pshare}(\mathbf{r}^{q,\mathsf{right}}, \mathcal{A}, [\mathbf{u}^{q,\mathsf{right}}]_{\mathcal{A}}, D)$.
- It hands each honest party P_i its shares $\{\mathbf{u}_i^{q,\mathsf{left}}, \mathbf{u}_i^{q,\mathsf{right}}\}_{q\in[w_{m+1,j}/\ell]}$ and $\{\mathbf{v}_i^{q,\mathsf{mult}}, \mathbf{v}_i^{q,\mathsf{add}}, \mathbf{v}_i^{q,\mathsf{relay}}\}_{q\in[w_{m,j}/\ell]}$.

Fig. 4: Random share generation functionality

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 5.

6.2 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 input x_i to functionality $f_{\mathsf{pack-input}}$ and records its vector of packed shares

[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.

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

Input: The parties $\{P_i\}_{i\in[n]}$ hold packed secret sharings $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}], [\mathbf{x}_2^{j,q}], [\mathbf{y}_2^{j,q}]\right\}_{j\in[\sigma], q\in[w_{m,j}/\ell]}$ and configuration of layers layer_m and layer_{m+1}.

Protocol: For each $j \in [\sigma]$, the parties proceed as follows:

- They invoke $f_{\mathsf{corr-rand}}$ to obtain packed secret shares: $\{[\mathbf{r}^{j,q,\mathsf{left}}], [\mathbf{r}^{j,q,\mathsf{right}}]\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle \mathbf{r}^{j,q,\mathsf{mult}}\rangle, \langle \mathbf{r}^{j,q,\mathsf{add}}\rangle, \langle \mathbf{r}^{j,q,\mathsf{relay}}\rangle\}_{q\in[w_{m,j}/\ell]}$.
- For each $q \in [w_{m,j}/\ell]$, the parties locally compute the following:
  $\langle \mathbf{x}_1^{j,q} \cdot \mathbf{y}_2^{j,q} + \mathbf{r}^{j,q,\mathsf{mult}}\rangle = [\mathbf{x}_1^{j,q}] \cdot [\mathbf{y}_2^{j,q}] + \langle \mathbf{r}^{j,q,\mathsf{mult}}\rangle$
  $\langle \mathbf{x}_1^{j,q} + \mathbf{y}_1^{j,q} + \mathbf{r}^{j,q,\mathsf{add}}\rangle = [\mathbf{x}_1^{j,q}] + [\mathbf{y}_1^{j,q}] + \langle \mathbf{r}^{j,q,\mathsf{add}}\rangle$
  $\langle \mathbf{x}_1^{j,q} + \mathbf{r}^{j,q,\mathsf{relay}}\rangle = [\mathbf{x}_1^{j,q}] + \langle \mathbf{r}^{j,q,\mathsf{relay}}\rangle$
- All the parties send their shares to the designated party P_{leader} for that layer.
- Party P_{leader} 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 $z_i^{j,1}, \ldots, z_i^{j,w_{m+1}}$ 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 a multiplication gate, it sets $z^{j,i} = z_i^{j,\mathsf{add}}$.
  - If gate $g_m^{j,i}$ is a relay gate, it sets $z^{j,i} = z_i^{j,\mathsf{relay}}$.
- It then computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.
- 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 $\mathbf{z}^{j,q,\mathsf{left}}[i] = z^{j,e_{\mathsf{left}}}$ and $\mathbf{z}^{j,q,\mathsf{right}}[i] = z^{j,e_{\mathsf{right}}}$.
- For each $j \in [\sigma], q \in [w_{m+1,j}/\ell]$, it then runs $\mathsf{pshare}(\mathbf{z}^{j,q,\mathsf{left}}, D)$ to obtain shares $[\mathbf{z}_i^{j,q,\mathsf{left}}]$ and $\mathsf{pshare}(\mathbf{z}^{j,q,\mathsf{right}}, D)$ to obtain a shares $[\mathbf{z}^{j,q,\mathsf{right}}]$ for each party.
- 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— $[\mathbf{z}^{j,q,\mathsf{left}}] = [\mathbf{z}^{j,q,\mathsf{left}}] - [\mathbf{r}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}] = [\mathbf{z}^{j,q,\mathsf{right}}] - [\mathbf{r}^{j,q,\mathsf{right}}]$.

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

Fig. 5: A Protocol for Secure Layer Evaluation

$\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j\in[\sigma], q\in[w_{1,j}/\ell]}$ 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 circuit evaluation proceeds layer-wise, where for each layer $m \in [d]$, where d is the depth of the circuit, the parties evaluate each gate in that layer simultaneously as follows — Given packed input shares $\left\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}]\right\}$ for $j \in [\sigma], q \in [w_{m,j}/\ell]$, the parties run π_{layer} on inputs $\mathsf{layer}_{m+1}, \mathsf{layer}_m, \left\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}], [\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma], q\in[w_{m,j}/\ell]}$. They record their shares in $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma], q\in[w_{m+1,j}/\ell]}$.
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 give a proof of security for this protocol in the full-version of our paper. 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$/\ell$) 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 (this is because in the semi-honest setting, we do not need to check if the shares were computed honestly). 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$/\ell$). 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 $\frac{5|\mathsf{C}|\epsilon(3\epsilon+4)}{4+\epsilon} + 4n|\mathcal{I}|$.

6.3 Maliciously Secure Protocol

In this section, we now describe a protocol that achieves security with abort against malicious corruptions. In addition to the sub-functionalities and protocol discussed in Section 6.1, this protocol makes use of the following additional functionalities, we defer their description to the full version of our paper due to space constraints:

— Functionality $f_{\mathsf{pack-rand}}$ is realised by a protocol that outputs packed secret sharings of random vectors. Because of our requirements, we assume that this functionality operates in two modes - the $\mathsf{independent}$ mode will generate packed sharings of vectors in which each element is independent and the $\mathsf{uniform}$ mode will generate packed sharing of vectors in which each element is the same random value.
— Functionality f_{mult} is realised by a protocol that multiplies 2 pack-secret shared vectors.
— Functionality $f_{\mathsf{checkZero}}$ takes a pack-shared vector as input and checks whether or not it correcponds to a 0 vector.

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 $q \in [w_{1,j}/\ell]$.

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 circuit evaluation proceeds layer-wise, where for each layer $m \in [d]$, where d is the depth of the circuit, the parties evaluate each gate in that layer simultaneously as follows:
   — The parties run π_{layer} on inputs $\mathsf{layer}_m, \mathsf{layer}_{m+1}$, $\left\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}], [\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ and obtain their respective shares in $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$.
   — The parties then run π_{layer} on inputs $\mathsf{layer}_m, \mathsf{layer}_{m+1}$, $\left\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}], [\mathbf{rz}_m^{j,q,\mathsf{left}}], [\mathbf{rz}_m^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ and obtain their respective shares in $\left\{[\mathbf{rz}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{rz}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$.
4. **Verification Step:** Each party does the following:
   (a) For each $m \in [d]$, $j \in [\sigma]$,$q \in [w_{m,j}/\ell]$, 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]

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

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

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

[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.

Due to space constraints, we defer the proof of security for this protocol to the full version of our paper. We note that the above protocol only works for circuits over large arithmetic fields. In the full version, 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, $n - t - 1 \approx n(\frac{1}{2} + \frac{2}{\epsilon})$ of these packed shares can be computed with $5n^2 + 5n(t+1)$ communication. Therefore, overall the total communication required to generate all the randomness is the following:

— Correlated randomness for evaluating the circuit on actual inputs: $\frac{|C|}{\frac{n}{\epsilon} \times n(\frac{1}{2}+\frac{2}{\epsilon})}\left(5n^2 + 5n^2\left(\frac{1}{2} - \frac{2}{\epsilon}\right)\right) = \frac{5\epsilon|\mathsf{C}|(3\epsilon-4)}{\epsilon+4}$.
— Correlated randomness for evaluating the circuit on randomized inputs: $\frac{5\epsilon|\mathsf{C}|(3\epsilon-4)}{\epsilon+4}$
— 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 $\frac{12\epsilon|\mathsf{C}|(3\epsilon-4)}{\epsilon+4} + 10|\mathsf{C}|\epsilon = \frac{46\epsilon^2|\mathsf{C}|-8\epsilon|\mathsf{C}|}{\epsilon+4}$. 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 $\frac{46\epsilon^2|\mathsf{C}|-8\epsilon|\mathsf{C}|}{\epsilon+4} + 2n^2|\mathcal{I}|$.

7 Implementation and Evaluation

7.1 Theoretical Comparison to Prior Work

We start by comparing the concrete efficiency of our protocol based on the calculations from Section 6.3, where we show that the total communication complexity of our maliciously secure protocol is $\frac{46\epsilon^2|\mathsf{C}|-8\epsilon|\mathsf{C}|}{\epsilon+4} + 2n^2|\mathcal{I}|$. 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 [16] 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 $3\frac{1}{4}$ field elements per gate, in expectation. Notice that while we require parties to communicate for every gate in the circuit, [16] only requires communication per multiplication gate. However, it is easy to see that for circuits with approximately 75% multiplication gates, our protocol is expected (in theory) to outperform [16] for 125 parties.

Table 2: Comparing the runtime of our protocol and that of related work. Results for our circuits are reported for the average protocol execution time over five randomized circuits each with 1,000,000 gates. All times are rounded to seconds due to space constraints. Asterisk denote extrapolated runtimes between LAN setting and WAN setting (see text). On the right side of the table, prior work does not run for this number of parties, so we only include our own results.

Configuration			**Number of Parties**								
Net. Config	t	Depth	30	50	70	90	110	150	200	250	300
LAN (our work)	$n/4$	1000	29	37	49	53	60	-	-	-	-
LAN (our work)	$n/3$	1000	29	41	55	54	63	-	-	-	-
[7]	$n/2$	1000	12	26	33	49	80	-	-	-	-
[16]	$n/3$	20	1	2	3	4	> 4	-	-	-	-
WAN (our work)	$n/4$	1000	261	206	187	278	271	282	263	302	336
WAN (our work)	$n/3$	1000	299	285	215	261	305	315	279	320	378
[7]	$n/2$	20	87	128	164*	204*	257*	-	-	-	-
[7]	$n/2$	100	135	197	251*	355*	478*	-	-	-	-
[7]	$n/2$	1000	376*	816*	1k*	1.5k*	2.4k*	-	-	-	-

The advantage of $O(|\mathsf{C}|)$ protocols is that the per-party communication decreases as the number of parties increases. For the same corruption threshold of $t < n/3$, and $n = 150$ parties, our protocol would (on paper) only require each party to communicate $2\frac{2}{3}$ field elements per gate. In this case, our protocol is already expected to perform better than [16] for circuits that have more that 60% multiplication gates. As the number of parties increase, less of the circuit must be comprised of multiplication gates in order to show improvements. Alternatively, because our communication complexity depends on ϵ (that is directly proportional to the corruption threshold t), our protocol outperforms prior work with fewer parties if we reduce the corruption threshold. or $t < n/4$ corruptions and $n = 100$ parties, we require per-party communication of $2\frac{2}{5}$ field elements per gate.

Finally, we remark that the above is a theoretical comparison, and assumes the "best-case scenario", e.g., where the circuit is such that it has exactly $n-t-1$ 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, and some of the generated randomness will be "wasted" or some packed secret sharings will not be completely filled.

7.2 Implementation Comparison to Prior Work

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.

We implemented our maliciously secure protocol from Section 6.3. Additional details about our implementation can be found in the full version of the paper.

Our implementation is in C++ and built on top of libscapi [8], which provides communication and circuit parsing. To evaluate our implementation, we generate random layered circuits that satisfy the highly repetitive structural requirements. Benchmarking on random circuits is common, accepted practice for honest majority protocols [7,16]. We also modify the libscapi [8] circuit file format to allow for more succinct representation of highly repetitive circuits.

We ran tests in two network deployments, LAN and WAN. In our LAN deployment, all parties were co-located on a single, large server with two Intel(R) Xeon(R) CPU E5-2695 @ 2.10 GHz. In our WAN 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 instances 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 test 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 compare the runtime of our protocol in both our LAN deployment and WAN deployment to [7,16] 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 [7]. If we assume [16] 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 [7]. 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 left side of Table 2). For larger numbers of parties (see right side Table 2), where we have no comparable results, there is an upward trend in execution time. This could be a result of networking overhead or varying levels of network congestion when

[10] The only protocol to be run on large numbers of parties rests on incomparable assumptions like CRS [38].

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.*

8 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. 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

1. Fips pub 180-2, secure hash standard (shs), 2002. U.S.Department of Commerce/National Institute of Standards and Technology.
2. S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, Oct. / Nov. 2017.
3. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.
4. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.
5. M. Chase, D. Derler, S. Goldfeder, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, and G. Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, Oct. / Nov. 2017.
6. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In C. Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Heidelberg, Aug. 1988.

7. K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, Aug. 2018.
8. Cryptobiu. cryptobiu/libscapi, May 2019.
9. I. Damgård and Y. Ishai. Scalable secure multiparty computation. In C. Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, Aug. 2006.
10. I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In H. Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg, May / June 2010.
11. I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In D. Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 241–261. Springer, Heidelberg, Aug. 2008.
12. I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In A. Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, Aug. 2007.
13. R. Dingledine, N. Mathewson, and P. 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.
14. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, Aug. 1987.
15. M. K. Franklin and M. Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.
16. J. Furukawa and Y. Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 1557–1571. ACM Press, Nov. 2019.
17. P. Gallagher, D. D. Foreword, and C. F. 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.
18. A. Gascon, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Secure linear regression on vertically partitioned datasets. Cryptology ePrint Archive, Report 2016/892, 2016. http://eprint.iacr.org/2016/892.
19. D. Genkin. *Secure Computation in Hostile Environments*. PhD thesis, Technion - Israel Institute of Technology, 2016.
20. D. Genkin, Y. Ishai, and A. Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, Aug. 2015.
21. D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. Circuits resilient to additive attacks with applications to secure computation. In D. B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.
22. I. Giacomelli, J. Madsen, and C. Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In T. Holz and S. Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, Aug. 2016.

23. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In A. Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
24. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.
25. V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority MPC. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 618–646. Springer, Heidelberg, Aug. 2020.
26. M. Hastings, B. Hemenway, D. Noble, and S. 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.
27. M. Hirt and J. B. Nielsen. Robust multiparty computation with linear communication complexity. In C. Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 463–482. Springer, Heidelberg, Aug. 2006.
28. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In D. S. Johnson and U. Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.
29. J. Katz, V. Kolesnikov, and X. Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, Oct. 2018.
30. Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, Oct. / Nov. 2017.
31. P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, Oct. 2018.
32. P. Mohassel and Y. 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.
33. B. Mood, D. Gupta, H. Carter, K. Butler, and P. 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.
34. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. 2008.
35. P. S. Nordholt and M. Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In B. Preneel and F. Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 321–339. Springer, Heidelberg, July 2018.
36. 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.
37. A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, Nov. 1979.
38. R. Wails, A. Johnson, D. Starin, A. Yerukhimovich, and S. D. Gordon. Stormy: Statistics in tor by measuring securely. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, Nov. 2019.
39. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, Oct. 1986.