Generalized Pseudorandom Secret Sharing and Efficient Straggler-Resilient Secure Computation*

Fabrice Benhamouda
Algorand Foundation

Elette Boyle
IDC Herzliya, Israel

Niv Gilboa
Ben-Gurion University, Israel

Shai Halevi
Algorand Foundation

Yuval Ishai
Technion, Israel

Ariel Nof
Technion, Israel

November 14, 2021

Abstract

Secure multiparty computation (MPC) enables n parties, of which up to t may be corrupted, to perform joint computations on their private inputs while revealing only the outputs. Optimizing the asymptotic and concrete costs of MPC protocols has become an important line of research. Much of this research focuses on the setting of an honest majority, where $n \geq 2t + 1$, which gives rise to concretely efficient protocols that are either information-theoretic or make a black-box use of symmetric cryptography. Efficiency can be further improved in the case of a *strong* honest majority, where $n > 2t + 1$.

Motivated by the goal of minimizing the communication and latency costs of MPC with a strong honest majority, we make two related contributions.

- **Generalized pseudorandom secret sharing (PRSS).** Linear correlations serve as an important resource for MPC protocols and beyond. PRSS enables secure generation of many pseudorandom instances of such correlations without interaction, given replicated seeds of a pseudorandom function. We extend the PRSS technique of Cramer et al. (TCC 2005) for sharing degree-d polynomials to new constructions leveraging a particular class of combinatorial designs. Our constructions yield a dramatic efficiency improvement when the degree d is higher than the security threshold t, not only for standard degree-d correlations but also for several useful generalizations. In particular, correlations for locally converting between slot configurations in "share packing" enable us to avoid the concrete overhead of prior works.
- **Cheap straggler resilience.** In reality, communication is not fully synchronous: protocol executions suffer from variance in communication delays and occasional node or message-delivery failures. We explore the benefits of PRSS-based MPC with a strong honest majority toward robustness against such failures, in turn yielding improved latency delays. In doing so we develop a novel technique for defending against a subtle "double-dipping" attack, which applies to the best existing protocols, with almost no extra cost in communication or rounds.

Combining the above tools requires further work, including new methods for batch verification via distributed zero-knowledge proofs (Boneh et al., CRYPTO 2019) that apply to packed secret sharing. Overall, our work demonstrates new advantages of the strong honest majority setting, and introduces new tools—in particular, generalized PRSS—that we believe will be of independent use within other cryptographic applications.

*This paper is the full version of [7]

Contents

1 Introduction

Protocols for secure multiparty computation (MPC) [59, 34, 5, 17] enable a set of parties with private inputs to compute a joint function of their inputs while revealing nothing but the output. MPC provides a general-purpose tool for distributed computation on sensitive data, as well as for eliminating single points of failure. As a result, a major research effort focused on improving the asymptotic and concrete efficiency of MPC.

Efficient honest-majority MPC. The most practical MPC protocols rely on an *honest majority* assumption, namely security is guaranteed as long as $t < n/2$ out of the n parties are corrupted, and provide "security with abort" in the presence of malicious parties. Such protocols can be either information-theoretic, or alternatively achieve better communication cost by making a black-box use of a pseudorandom function. The latter is mainly useful for non-interactive generation of pseudorandom shared secrets via a pseudorandom secret sharing (PRSS) technique [32, 21]. Moreover, the most efficient protocols in this setting follow the blueprint of Damgård and Nielsen (DN) [25], where each layer of a circuit is evaluated by having a designated "leader" party send messages to all other parties and receive a message from each party in return.

In almost all of this line of research, one assumes the weakest honest majority assumption of $n = 2t+1$ parties. However, assuming that up to half of the parties can be corrupted may sometimes be overly *pessimistic*, and small relaxations of corruption threshold can be highly preferred in favor of boosting performance. On the other hand, existing honest-majority protocols are also overly *optimistic* in that they assume all messages arrive on time and are not robust to transient delays or failures. We will revisit this issue later.

The potential for savings in the "strong honest majority" regime of $n > 2t+1$ has been asserted within the context of *asymptotic* efficiency [27, 22, 24, 23, 4, 30, 41]. In a sense, existing MPC protocols for $n = 2t + 1$ parties are analogous to using a repetition code, which increases the total cost by a factor of n, whereas the latter protocols are analogous to asymptotically good codes that provide a constant or near-constant amortized asymptotic overhead. However, the techniques in these theory-oriented works incur large concrete overheads placing them quite far from practical efficiency, and their asymptotic efficiency benefits kick in only for large computations.

In the context of *concretely* efficient MPC, the potential gains of a strong honest majority remain relatively untapped—both in the sense that asymptotic benefits of prior works do not currently translate to concrete wins, and that potential for concrete gains outside the standard theoretical models or (asymptotic) goals have not been well explored. One exception to this is a recent line of works leveraging a larger number of honest parties for the purpose of closing the efficiency gap between security against *malicious* (or active) adversaries and security against *semi-honest* (or passive) adversaries [36, 29]. However, recent works [9, 12, 42, 13] have successfully closed this gap even given a minimal honest majority $n = 2t + 1$, in which case this advantage no longer applies.

In this work, we initiate a deeper study of *concretely efficient* MPC with *strong honest majority* $n > 2t + 1$. We focus on developing general-purpose primitives and techniques to alleviate the concrete costs of existing theory-oriented solutions, as well as exploring new directions for improved latency in realistic networks. Our primary focus is on the case where the corruption threshold t is small. This enables the use of PRSS techniques that give rise to simpler and more efficient protocols, but incur (an offline) cost that scales exponentially with t. We are motivated by two main limitations of current techniques.

The overhead of packed secret sharing. A major source of concrete overhead in the aforementioned theory-oriented works is the use of a "share packing" technique [27] in which secret-shared values are arranged into blocks, and a set of shares can simultaneously encode several values at the same per-party cost. This technique natively supports computing a single circuit on many inputs in parallel (also known as a "SIMD computation"), by computing operations simultaneously on all values within a block. However, it requires a costly routing mechanism for general computations. This overhead applies even in the semi-honest setting, but introduces additional challenges in the malicious setting. While the initial $O(\log n)$ overhead of the routing-based technique from [23] was recently improved to a constant [41], this comes at the cost of poor concrete efficiency.

Extending the ideas of these works, one may observe that existence of certain useful linear correlations across parties would enable avoiding these routing overheads altogether. The desired correlations correspond to sets of packed shares of secret random values, where different sets include the same random values in different computation "slot" positions, in line with the routing of wires within the computation circuit. But, unlocking these savings demands a large number of *different* rerouting patterns, whose generation would destroy the optimization savings in existing works. Much of the effort in previous works [22, 24, 23, 4, 38, 30, 41] was spent on efficient distributed protocols for generating these linear correlations.

Tolerating stragglers. One advantage of MPC with a strong honest majority, which serves as a primary motivation for the current work, is the potential for better robustness, in turn leading to *reduced latency* in realistic network environments. Existing MPC protocols with $n = 2t + 1$ parties require at least one of the parties to wait for messages from *all other parties* before proceeding to the next round. In particular, in protocols that follow the DN blueprint, the leader needs to wait until it hears back from all other parties. But in reality, communication is not fully synchronous. Even in a semi-honest setting, protocol executions suffer from variance in communication delays and occasional message-delivery failures. This is sometimes referred to as the problem of *stragglers*. To deal with this problem, practical distributed systems typically employ redundancy to allow proceeding with the computation as long as "sufficiently many" messages were received. See [48] for empirical studies of the impact of stragglers on realistic network.

Interestingly, achieving robustness to stragglers becomes more challenging when some parties can be malicious. Standard secure protocols with good concrete efficiency do not have this feature even when $n > 3t$. While such protocols are able to terminate in the face of up to t stragglers, this occurs at the cost of labeling these parties as corrupt, and their secrets are no longer protected. Alternatively, attempting to run DN-style protocols in an "optimistic mode," by simply having the leader wait for the first $2t$ messages to arrive, gives rise to a subtle "double-dipping" attack that allows a malicious leader to learn private information. Previous solutions for this attack(see [29, 40]) require significantly more interaction and are not suitable for efficiently dealing with transient faults; See Section 1.2 and Section 5.1 for more details.

1.1 Our Contributions

Motivated by the above opportunities and challenges, we present new techniques for MPC within the setting of a *strong honest majority*, $n > 2t + 1$, focusing on the case of small[1] values of t that

[1] More precisely, our protocols have storage and (offline) computation costs that grow exponentially in t but linearly in the number of parties n. Thus, when t is a small constant, they can be practical even for a large n.

enable efficient use of PRSS. We make the following two main contributions.

Contribution 1: Generalized pseudorandom secret sharing (PRSS). As noted above, PRSS enables a secure non-interactive generation of (pseudo)random values that are uniformly distributed over some linear vector space. It relies on a low-communication setup, where independent pseudorandom function (PRF) seeds are distributed to different subsets of the parties. The prominent cost metric of a PRSS scheme is the number of such seeds required for the parties to each compute their entry within the sampled vector. Following a general framework of Gilboa and Ishai [32], Cramer et al. [21] described PRSS techniques for sharing degree-d polynomials between n parties using $\binom{n}{d}$ seeds, $\binom{n-1}{d}$ per party, targeting the typical use-case where the security threshold t is equal to d. Motivated by the fact that in MPC with strong honest majority we have $t < d$, we present new PRSS constructions exploiting this gap.

Our constructions leverage suitable combinatorial designs, and yield a dramatic efficiency improvement when $t \ll d$, not only for standard degree-d correlations but also for several useful generalizations. This includes correlations for locally converting between slot configurations in "share packing," which enable us to avoid the concrete overhead of prior works on MPC based on share packing [23, 38, 30]. We remark that our PRSS results are independently motivated by other applications beyond the context of MPC, including threshold cryptography, advanced cryptographic primitives, and targeted multi-party protocols (e.g., [16, 26, 6, 14, 8]). See Appendix A for discussion.

We provide a general transformation yielding PRSS schemes from any instance of a so-called "covering design" with appropriate parameters. An (n, m, t)-*cover* is a collection of size-m subsets $S_i \subset [n]$, such that every subset of t elements of $[n]$ is covered by some set S_i. The goal is to do so with the fewest number of such sets S_i. Construction of covering designs is a topic of combinatorial research, where bounds are known for small parameters, and several results are known in the larger parameter regime (see Section 3.3 for discussion). While it is not hard to see that the seed replication pattern of a PRSS must induce a covering design, the converse direction is less obvious. Indeed, our transformation incurs a small overhead that leaves a $(d+1)$ multiplicative gap between the upper and lower bounds on the number of seeds for the case of degree-d polynomials.

The following theorem summarizes our general transformation from covering designs to PRSS for degree-d polynomials, as well as some corollaries obtained by plugging in existing covering designs from the literature (cf. [37]).

Theorem 1.1 (PRSS for degree-d polynomials from covering designs, informal) *Let n, d, t be positive integers such that $t < d < n$. Given an $(n, d+1, t)$-cover of size k, one can construct a PRSS scheme for sharing random degree-d polynomials between n parties with security threshold t, requiring $k(d+1)(n-d)/n$ PRF seeds per party. As a special case, plugging in existing covering designs for small t, we obtain the following:*

- *For $t = 1$, any n: $\left\lceil \frac{n}{d+1} \right\rceil \frac{(d+1)(n-d)}{n}$ PRF seeds per party (or just $n - d$ when $(d+1)|n$); .*
- *For $t = 2$, any $n \leq 3(d+1)$: $13(d+1)$ PRF seeds per party.*

We further obtain PRSS for "double Shamir sharing" (i.e. two random polynomials of degrees d and $2d$ with the same evaluations on given $d-t+1$ points) with roughly twice as many PRF seeds.

In comparison to the parameters above, the naive baseline from [21] is $\binom{n-1}{d}$ seeds per party, which in the case that $t < d$ can be improved to $\binom{n-1}{t}$ seeds per party (see Footnote 2). Plugging in explicit covering design constructions from the literature, the PRSS solutions obtained via Theorem 1.1 provide significant savings to even this improved baseline. For example:

- $(n, d, t) = (48, 15, 4)$ requires $2,772$ seeds per party, versus baseline $\binom{47}{4} = 178,365$.
- $(n, d, t) = (49, 23, 4)$ requires 484 seeds per party, versus baseline $\binom{48}{4} = 194,580$.
- $(n, d, t) = (49, 23, 8)$ requires $57,281$ seeds per party, versus baseline $\binom{48}{8} \approx 3.7 \cdot 10^8$.

See Table 1 for additional data points. Our PRSS constructions go beyond basic Shamir or double-Shamir shares, to a generalized form of PRSS that allows local generation of packed pseudorandom secrets with an arbitrary replication pattern. We achieve this with with additional redundancy of seeds to parties. However, the resulting complexity still provides significant savings as an alternative to existing approaches within motivated regimes. We refer the reader to Section 3.6 for a detailed treatment.

Contribution 2: Cheap straggler resilience. We propose a novel technique for dealing with the "straggler" problem of delayed messages, allowing the protocol to continue the execution once sufficiently many messages are received. In doing so, we need to defend against the subtle "double-dipping" attack mentioned above. In contrast to alternative approaches for defending against this attack [29, 40], our approach has no extra cost to the round complexity of the protocol and only a sublinear additive communication overhead. Our protocol makes black-box use of our PRSS construction to produce the required randomness without interaction.

Combining the above tools to obtain efficient MPC protocols with security against malicious parties requires additional ideas. In particular, we need to adapt the distributed zero-knowledge proof techniques of Boneh et al. [9] to the setting of MPC based on packed secret sharing. See additional discussion below.

The features of our final protocol are captured by the following theorem.

Theorem 1.2 (Malicious security with straggler resilience, informal) *Let $t \geq 1$ be a security threshold, $\ell \geq 1$ a packing parameter, $n \geq 2t + 2\ell - 1$ a number of parties, and $\mathbb{F}$ be a finite field such that $|\mathbb{F}| > n + t + 2\ell$. Then, for any arithmetic circuit C over $\mathbb{F}$ with S multiplication gates and depth D, there is an n-party protocol for C with the following efficiency and security features:*

- *The protocol makes a black-box use of any pseudorandom function;*
- *Excluding $O(1)$ rounds of preprocessing and postprocessing, the protocol consists of D epochs, where in each epoch P_1 sends a message to each other party P_i and receives a message back from each P_i;*
- *It achieves security with abort in presence of t malicious parties even if $\tau = n - (2t + 2\ell - 1)$ messages, chosen by the adversary, are dropped in each epoch;*
- *If the parties follow the protocol, it terminates successfully even if τ messages, chosen by the adversary, are dropped in each epoch;*

- *Communication cost is* $\left(\frac{3}{\ell} - \frac{2t+2\ell+1}{n \cdot \ell}\right) S + o(S)$ *elements of* $\mathbb{F}$ *sent per party.*

We further discuss the communication, computation, and storage costs in the following remarks.

Remark 1.3 (Sensitivity to the topology of C.) *As in other protocols based on packed secret sharing, the communication complexity bound in Theorem 1.2 assumes that the circuit C is "non-pathological" in the sense that its width is bigger than the packing parameter ℓ. (Otherwise there is an extra communication cost resulting from empty slots.) Since we typically expect ℓ to be much smaller than the circuit size, this condition is met for almost all natural instances of big circuits.*

Remark 1.4 (On the cost of PRSS.) *The generalized PRSS primitive influences the local storage and computational cost, which can be performed offline and are practical for small t even for large values of ℓ and n; see Table 1 and Table 3 for concrete numbers. By increasing the degree parameter d of the generalized PRSS construction beyond the minimum required by t and ℓ, we get better PRSS complexity at the cost of a lower straggler resilience threshold τ.*

Remark 1.5 (On communication complexity.) *When $\ell = 2$, the amortized communication cost in Theorem 1.2 is always less than 1.5 elements per party per gate, and when $\ell = 3$ it goes below 1 element per party. We present concrete efficiency analysis of our protocol in Table 2, showing that as we increase n and ℓ, our protocol not only can withstand stragglers, but also achieves lower total communication than the best known semi-honest protocols with $n = 2t + 1$ parties. In particular, whenever $\ell = \Omega(n)$ the total communication complexity (ignoring lower order additive terms) is $O(s)$.*

Technical challenges & contributions. Our final MPC protocol builds on new solutions for the following main challenges:

- *Generalized pseudorandom secret sharing (PRSS)* based on combinatorial designs that take advantage of the gap between the polynomial degree d and the security threshold t to reduce computation and storage costs.
- *Packed secret sharing beyond SIMD*, without the asymptotic or concrete overhead of previous techniques [23, 38, 30]. Our solution relies on generalized PRSS for cheaper batch generation of useful linear correlations, for "repacking" secret shared values in different orders.
- *Preventing "double-dipping" attacks*, identified by [40, 29], which exploit the redundancy of encoding across parties in a strong honest majority to obtain related secret values under the same random mask (see below; note that this attack arises even without share packing). The works of [40, 29] protect against the attack using methods that require participation from *all* parties and increase the round complexity by 2x or more; we do so while supporting resilience to stragglers, and with essentially no extra online cost.
- *Applying sublinear distributed zero knowledge [9] on packed shares*, as well as achieving batched verification with missing shares (due to stragglers). The former challenge arises again from the non-SIMD structure of general computation, here relating to the statements to be efficiently *verified*. The latter issue pertains to verifying consistency of several robustly secret shared values, when each secret has a *different* subset of shares missing, corresponding to different sets of straggling parties.

We further discuss each of the above points in Section 1.2 below.

1.2 Overview of Techniques

Generating random packed sharings via generalized PRSS. In section 3 we show how to generate shared blocks of random sharings that satisfy linear correlations, without any interaction beyond a short setup step. Our techniques can be seen as an extension of the share conversion methods of Cramer-Damgård-Ishai (CDI) [21]. However, besides extend this construction, we also drastically improve its efficiency.

Recall that CDI described non-interactive sharing of (pseudo)random degree-t polynomials, resilient against t dishonest parties, as follows: Each $(n-t)$-subset of parties $S \subset [n]$ (we use $[n]$ to denote the set $\{1,\ldots,n\}$) gets a random PRF seed, and uses it to generate a (pseudo)random degree-t polynomial which is zero in all points outside of S. The resulting polynomial is just a sum of all these $\binom{n}{t}$ polynomials. Each party can compute its share on this polynomial from the seeds that it knows, because the seeds that it is missing correspond to polynomials that evaluate to zero in the point of that party.

We extend this construction to the case where the polynomial degree d is larger than the resilience threshold t, and show how the gap between d and t can be used to achieve drastic improvement in complexity. In this case, the construction turns out to be closely related to a well studied combinatorial design problem of covering all t-subsets of $[n]$ using larger subsets of size (roughly) d. That is, each t-subset of $[n]$ must be contained completely within at least one d-subset within the cover.

It is easy to see that such covers are necessary: Each PRF seed must be given to a size-$(n-d)$ subset for the resulting polynomial to have degree d (since any party without the seed must correspond to evaluation output 0), and every t-subset must miss at least one of the seeds (else they know the entire polynomial). So the complementing subsets must form a set-cover of all t-subsets using d-subsets. We observe that the other direction also holds, albeit with some overhead: Any design that covers all t-subsets using larger subsets of size $d+1$ can be converted to a secure share-conversion for degree-d polynomials with security against t-collusions, with only a modest increase in complexity.

Specifically, let $\mathcal{C}' = \{S'_1, S'_2, \ldots, S'_{k'}\}$ be a collection of size-$(d+1)$ subsets that covers all t-subsets (i.e., for every t-subset T there exists $S'_i \in \mathcal{C}'$ such that $T \subset S'_i$.) Consider all the subsets that are obtained by removing a single element from any of the S'_i's, namely the collection

$$\bar{\mathcal{C}} = \big\{S' \setminus \{j\} : \ S' \in \mathcal{C}', j \in S'\big\}.$$

The number of distinct subsets in $\bar{\mathcal{C}}$ is $k \leq k'(d+1)$, and each subset is of size d. Let us denote the subsets in $\bar{\mathcal{C}}$ by $\bar{S}_1, \bar{S}_2, \ldots, \bar{S}_k$, and their complement sets by $S_i = [n] \setminus \bar{S}_i$. We use the S_i's in the CDI construction to distribute (pseudo)random polynomials. The resulting polynomials are of degree d (since the S_i's have cardinality $n-d$). It is also easy to see that $\bar{\mathcal{C}}$ still covers all t-subsets (and hence each t-subset still misses some seeds): For each $T \subset [n]$ there is $S'_i \in \mathcal{C}'$ that covers it, and removing from S'_i an element *which is not in* T yields some $\bar{S} \in \bar{\mathcal{C}}$ that still covers T. In section 3 we use the linear algebraic security criteria of Gilboa and Ishai [32] to prove that this construction is indeed resilient against any t-collusion. We then show a more intricate extension of this construction that handles the more complex linear correlations corresponding to packed random secrets that satisfy an arbitrary replication pattern. These are useful for our MPC protocol which we describe below.

The key reason for the efficiency advantage of generalized PRSS over standard PRSS is that when $d \gg t$ there are set-covers of size much less than $\binom{n}{t}$, sometimes as small as $O(1)$ (when

$t = O(1)$ and $d = \Omega(n)$). In these cases we get constructions with total complexity $O(n)$, compared to $O(n^t)$ of standard PRSS [21].[2] See Table 1 for concrete numbers.

First step toward straggler resilience. We proceed to discuss our protocol which deals with stragglers. The starting point is the Damgård-Nielsen (DN) [25] protocol for computing the multiplication of two shared values x and y. This protocol is the most efficient honest-majority secure multiplication protocol known to date. We use the notation $[\![x]\!]_d$ to denote a Shamir's secret sharing of x via a degree-d polynomial. In the DN protocol, the parties prepare random sharings $[\![r]\!]_d$ and $[\![r]\!]_{2d}$ which are used as follows. To multiply $[\![x]\!]_d$ and $[\![y]\!]_d$, the parties locally multiply their shares, mask it by adding $[\![r]\!]_{2d}$ and send the result to a designated party P_1. Party P_1 then reconstructs $xy - r$ and shares it to the parties as $[\![xy - r]\!]_d$. Finally, the parties locally add $[\![xy - r]\!]_d$ to $[\![r]\!]_d$ to obtain $[\![xy]\!]_d$. The sharing $[\![r]\!]_{2d}$ is a random masking polynomial, which guarantees that no private data is learned (since locally multiplying $[\![x]\!]_d$ and $[\![y]\!]_d$ yields $[\![xy]\!]_{2d}$, the masking polynomial must be of degree at least $2d$).

It is easy to see that P_1 must receive $2d$ messages in order to compute $xy-r$, as together with its own share, P_1 now holds $2d+1$ points on a $2d$-degree polynomial. Thus, in the semi-honest setting, a straightforward solution to withstand loss of messages in this step is to consider additional parties, i.e., strong honest majority. Given $n > 2d + 1$ parties, the protocol directly becomes resilient to $n - (2d + 1)$ dropped messages in a phase consisting of the parties sending degree-$2d$ shares to P_1 and receiving degree-d shares in return.

Reducing communication and computation using packed secret sharing. One drawback of increasing the number of parties, is that the overall communication and computation grow as well. To reduce communication and to allow straggler resilience without increasing costs, we wish to leverage the ideas of [22, 23] using packed secret sharing [27], to allow encoding a block of ℓ secrets on the same polynomial. We use the notation $[\![x_1 \cdots x_\ell]\!]_d$ to denote that the secrets $x_1, \ldots, x_\ell$ are shared via a polynomial of degree d. Following [22, 23], given two blocks $[\![x_1 \cdots x_\ell]\!]_d$ and $[\![y_1 \cdots y_\ell]\!]_d$, and randomness of the form $[\![r_1 \cdots r_\ell]\!]_d$ and $[\![r_1 \cdots r_\ell]\!]_{2d}$, it is possible to invoke the DN protocol *once* to obtain $[\![x_1 y_1 \cdots x_\ell y_\ell]\!]_d$, instead of calling it ℓ times in the single secret-per-sharing case, thereby reducing costs by a factor of ℓ.

This approach fits directly into place when the program to be evaluated is of a special SIMD ("same instruction, multiple data") structure, consisting of many parallel copies of an identical sub-computation. However, as encountered in these works, a greater challenge comes in supporting computations of general circuit structures *beyond SIMD*, since outputs of intermediate operations must be reordered into different "slots" to perform the following block of parallel operations. This problem was addressed in prior works by introducing between each pair of existing operations an additional logarithmic sequence of intermediate "routing" operations (emulating the swaps of a routing network), which serve to reorder the outputs into new blocks that correspond to the program structure. However, this routing procedure is the source of great overhead. Asymptotically it

[2] While a naive use of CDI [21] for degree-d polynomials requires $\binom{n}{d}$ PRF seeds, when $t < d$ it can be reduced to $\binom{n}{t}$ by using the same seed multiple times, each time with a different input. Concretely, for each set of parties of size T, where $|T| = t$, a seed r_T is given to the parties in $[n] \setminus T$. The parties define a polynomial P_T where all points corresponding to the parties in T are zero, and $d+1-t$ additional points are set by invoking the PRF, with r_T as the key, and the point's x-coordinate as the input (e.g., for each $i \in T$ set $P_T(i) = 0$, and for each $k \in \{-d+t, \ldots, 0\}$ set $P_T(k) = PRF_{r_T}(k)$). Note that the parties in $[n] \setminus T$ have enough information to compute P_T. The final polynomial of degree d is the sum of all these $\binom{n}{t}$ polynomials.

multiplies the complexity of the protocol by a logarithmic factor, and concretely it incurs significant slowdowns and extra implementation complexity.

One may observe that, given a particular form of linear correlated randomness, a simple adjustment to the existing protocols can allow the use of packing techniques in general circuits/programs *without increasing communication*. We can additionally leverage the fact that in the DN protocol, party P_1 sees masked intermediate values in the clear, to let P_1 carry out locally linear operations over the masked secrets.

For example, consider $\ell = 2$ and suppose that P_1 received $[\![x_1 + r_1, x_2 + r_2]\!]_{2d}$ in some intermediate layer of the circuit and recovered the values x_1+r_1 and x_2+r_2 in the clear. Further suppose that for the next layer of the circuit, x_1 is supposed to be packed in both positions of some block. P_1 can distribute to everyone the packed polynomial $[\![x_1 + r_1, x_1 + r_1]\!]_d$, and it falls to the offline randomness generation to equip the parties with a sharing of $[\![r_1, r_1]\!]_d$ that they can use to unmask these values. Similarly, suppose that P_1 holds x_1+r_1 and x_2+r_2, and the circuit specifies that some linear combination $\alpha_1 \cdot x_1+\alpha_2 \cdot x_2$ is to be fed into a multiplication gate in the next level, packed in the first position of some block. P_1 can compute $x' = \alpha_1(x_1+r_1)+\alpha_2(x_2+r_2) = (\alpha_1 x_1+\alpha_2 x_2)+(\alpha_1 r_1+\alpha_2 r_2)$, and distribute a sharing of $[\![x', \ldots]\!]_d$ to everyone (with some other value in the second position). Again it falls to the offline randomness generation to equip the parties with a sharing of $[\![r', \ldots]\!]_d$ (with $r' = \alpha_1 r_1 + \alpha_2 r_2$) that they can use for unmasking.

In full generality, we can implement an arbitrary linear transformation between two adjacent multiplication layers by having P_1 apply this linear transformation to the masked values, and having the offline randomness generation equip the parties with the linearly correlated randomness needed for unmasking. Specifically, the parties must hold blocks of random secrets $[\![r_1 \cdots r_\ell]\!]_{2d}$ for masking and $[\![r'_1 \cdots r'_\ell]\!]_d$ for unmasking, such that the r_i's are uniformly random and each r'_j is set as some public linear combination of the r_i's. Fortunately, this type of correlated randomness can be provided by our PRSS method discussed above as we show in Section 3.6.

From semi-honest to malicious security. The next step is to augment the protocol described so far to malicious security (with abort). Our goal is to achieve this without increasing the amortized communication cost, and while providing the same resilience to stragglers as in the semi-honest protocol. As in many efficient multi-party protocols, we follow the path of letting the parties run the semi-honest protocol to compute the circuit/program, and then, before the output is revealed, run a short cheap verification protocol to detect cheating in the semi-honest protocol. However, this raises a problem.

While semi-honest protocols (and in particular the DN protocol) in the setting of $n = 2d+1$ have been shown to achieve privacy even in the presence of malicious adversaries, surprisingly enough this is *not* the case when $n > 2d+1$, as considered in this work. Indeed, Goyal et al. [40] identified a subtle yet fatal concrete attack (which we call the "double-dipping" attack) for this setting that is carried out over two multiplication gates in two layers, in which a malicious P_1 can learn private data. What enables this attack is the fact that P_1 needs only $2d+1$ shares to compute $xy-r$, and can use these shares to also compute by himself the remaining $n-(2d+1)$ shares (since any $2d+1$ points on a $2d$-degree polynomial determine deterministically all the other points). This enables P_1 to send a false message to a single party and then later compare the "correct" response of this party in the following round (computed based on the other parties' correct response) to the party's real response (computed based on the false message sent by P_1 in the previous round), revealing secret-dependent information.

To overcome this problem, a simple solution suggested in [40] is to take a masking polynomial of degree $n-1$ instead of $2d$. A different solution from [29] requires running a constant-cost check between each two layers of the computed circuit/program. The first solution require *all* parties to participate, thus leaving no room for stragglers, while the second blows up the number of rounds by a factor of 2.

Privacy against "double-dipping" attacks. To allow straggler resilience without changing the round complexity, we design a new solution that allows P_1 to proceed with just $2d+1$ shares as before, but in a private manner. The main idea behind our solution is to have a *different* masking polynomial for each subset of $2d+1$ parties. This of course raises the question of which masking share a party should use when sending its message to P_1. Our idea thus requires that each subset T of $2d+1$ parties will hold a sharing $[\![r_T]\!]_{2d}$, under the constraint that *the share held by each party P_i will be the same for all subsets T for which $P_i \in T$*. The protocol then proceeds by having each party use its share as a masking in its message to P_1, and having the exact masking polynomial be determined "on the fly" by the first $2d$ messages arriving to P_1. This method achieves privacy since even though the masking polynomial is of degree $2d < n-1$, the different masks mean that the random shares held by the parties are now completely random and independent (i.e., cannot be reconstructed from one another), preventing the above attack.

This new method requires of course the adjustment of the offline protocol described above to produce random sharings with these constraints. We show how these can be produced to support our new protocol using our general PRSS method in a black-box way, enabling a private evaluation of the circuit/program in the presence of malicious adversaries with resilience to $n-(2d+1)$ dropped messages in each phase as before.

Sublinear zero knowledge on packed shares (and batched verification with stragglers). Once privacy is guaranteed, we proceed to show how to achieve correctness. To this end, we utilize the distributed zero-knowledge proofs introduced by Boneh et al. [9]. Their main observation is that to verify the correctness of a distributed computation (as opposed to carrying out the computation), one can define a verification circuit of degree-2, where all the inputs are robustly shared among the parties, which outputs 0 if no cheating took place. Specifically, given that the parties hold multiplication triples $([\![x]\!]_d, [\![y]\!]_d, [\![z]\!]_d)$ corresponding to the inputs and output of each multiplication gate/instruction, the circuit that takes a random linear combination of all the differences of the form $z - x \cdot y$ is a degree-2 circuit with inputs that are shared robustly via degree-d sharings and that outputs 0 if all multiplication triples are correct. The low degree of the circuit enables running a secure protocol to verify that the output is 0, with sublinear communication cost (in the size of the original circuit/program).

Applying their method to our setting raises several challenges. While the inputs to the verification circuit are robustly shared, they are encoded at different slots, which makes it difficult to perform the operations required by the protocol. We show how to overcome this by carefully designing a verification circuit, where all operation are slot-friendly operations, meaning that all operations are carried out over inputs that are encoded at the same entry of their block.

A second problem that arises is that in our protocol, the parties do not hold shares of the output of each multiplication. Recall that in our protocol P_1 does not share the output of each (masked) multiplication's output to the parties, but rather performs linear operations over the masked values and only when P_1 reaches the next multiplication, it shares the masked input blocks

to the parties. This seems problematic, since the verification circuit defined above works with multiplication triples. Nevertheless, we leverage the fact that the input to each multiplication is a linear combination of previous multiplications outputs, with coefficients that are publicly known, to construct a verification circuit of degree-2. We then show how the parties can ensure, by applying the mechanism of [9] on this verification circuit, that if cheating took place in one of the semi-honest computations, then it will be detected with high-probability. We note that besides the desired property of having sub-linear communication, our verification protocol is also constant-round, thereby not increasing the round complexity of the computation.

A final issue that arises is how to run the verification protocol, given that in the private protocol a subset of messages were dropped, which means that sharings of inputs to the verification circuit are only known to different subsets of parties. We observe that the number of such subsets is bounded by the depth of the program. Thus, we can run the lightweight verification a number of times that is bounded by $\min(\mathsf{depth}, \binom{n}{\tau})$ (where τ is the number of stragglers), obtaining communication that (for most natural programs) is still sublinear in the size of the program.

1.3 Related Work

We mention here specific recent works relating to our second contribution, of MPC in the strong honest majority setting achieving concrete efficiency and straggler resilience.

PRSS-based vs. interactive correlated randomness generation. In this work, we use non-interactive PRSS to generate the double sharing required for the DN multiplication protocol. While we improve the efficiency of PRSS dramatically (when the polynomial degree d is higher than the corruption threshold t), the computational overhead still limits the practical use of this method to a relatively small number of corrupted parties t. See Table 3 in Section 4 for concrete estimates of computational cost. An alternative to the PRSS-based approach is using an interactive protocol, but with computation that scales polynomially with the number of parties. The state-of-the-art protocol by Goyal *et al.* [39] shows how to generate the double sharing with communication of just 0.5 field element sent per party. This implies that our method requires approximately 25% less overall communication. More importantly, the method of [39] does not support straggler resilience and applies only to gate-by-gate evaluations. While it can be easily extended to SIMD circuits, it does not extend to general non-SIMD circuits with packed secrets. Finally, the correlated randomness generation procedure from [39] requires interaction between all parties, which can be prohibitive in some of the applications scenarios described in Appendix A.

MPC with strong honest majority. Concretely efficient MPC in the strong honest majority setting has gained recent focus, including the works of Gordon *et al.* [38] and Beck *et al.* [30]. In comparison, their protocols scale to a larger number of parties, while our approach provides better efficiency for the regime of small corruptions t. This is due largely to our ability to generate the necessary setup correlations with minimal interaction via generalized PRSS. In addition, our protocols provide straggler resilience (yielding savings in settings with latency variance), whereas [38, 30] assume a fully synchronous network. Finally, in these works, malicious security comes with a multiplicative overhead, whereas in our protocol, the overhead is sublinear in the size of the circuit.

A very recent work of Goyal *et al.* [41] shows how to achieve asymptotic constant communication cost per party in this setting for general non-SIMD circuits with information-theoretic security, but with poor concrete efficiency and without stragglers resilience.

MPC with partial synchrony. A number of works have studied MPC with various (stronger) flavors of partial synchrony from the perspective of feasibility, without focus on concrete efficiency. For example, the work of Zikas *et al.* [60] provides unconditionally secure protocols in a model where parties can additionally be send-omission or receive-omission corrupted. Guo *et al.* [43] consider a model where parties can periodically go offline and return. In Badrinarayanan *et al.* [3] parties can turn non-adversarially "lazy." Both of the latter rely on heavy cryptographic tools, such as (multi-key) fully homomorphic encryption.

Finally, a handful of works have considered concretely efficient MPC with forms of partial synchrony, with incomparable conclusions. Hirt and Maurer [47] consider a mixed model of malicious and fail-stop adversaries, achieve perfect security, but with larger overall cost (e.g., without the savings of share packing). The "Fluid MPC" work of Choudhuri *et al.* [19] builds efficient protocols within a very different model, designed for long computations, where in each period of time, a different set of parties carry-out the computation.

2 Preliminaries

Notation. Let $P_1, \ldots, P_n$ be the set of parties and let t, ℓ, d be integers such that $d \geq t+\ell-1$ and $n \geq 2d+1$. The parameter t bounds the number of parties that can be corrupted, the parameter ℓ denotes the size of the block of secrets that are evaluated together, and d will be the degree of the polynomial defined below. We use $[n]$ to denote the set $\{1, \ldots, n\}$ and denote by $\mathbb{F}$ a finite field.

2.1 Threshold Secret Sharing

Definition 2.1 *A d-out-of-n secret sharing scheme is a protocol for a* dealer *holding a secret value v and n parties $P_1, \ldots, P_n$. The scheme consists of two interactive algorithms:* $\mathsf{share}(v)$*, which outputs shares $(v_1, \ldots, v_n)$ and* $\mathsf{reconstruct}(\{v_j\}_{j\in T}, i)$*, which given the shares $v_j, j \in T \subseteq [n]$ outputs v or $\perp$. The dealer runs* $\mathsf{share}(v)$ *and provides P_i with a share v_i of the secret v. A subset of users T run* $\mathsf{reconstruct}(\{v_j\}_{j\in T}, i)$ *to reveal the secret to party P_i. The scheme must ensure that no subset of d shares provide any information on v, while $v =$* $\mathsf{reconstruct}(\{v_j\}_{j\in T}, i)$ *for T only if $|T| \geq d+1$. We say that a sharing is* $\mathsf{consistent}$ *if* $\mathsf{reconstruct}(\{v_j\}_{j\in T}, i) = \mathsf{reconstruct}(\{v_j\}_{j\in T'}, i)$ *for any two sets of honest parties $T, T' \subseteq \{1, \ldots, n\}$, and $|T|, |T'| \geq d+1$.*

2.1.1 Packed Shamir Secret Sharing

In Shamir's secret sharing scheme [55], the dealer defines a random polynomial $p(x)$ of degree d over a finite field $\mathbb{F}$ such that the constant term is the secret. Each party is associated with a distinct non-zero field element $\alpha \in \mathbb{F}$ and receives $p(\alpha)$ as its share of the secret. Since the degree of the polynomial is d, any $d+1$ points are sufficient to compute the secret. We use the notation $[\![x]\!]_d$ to denote a sharing of x via a polynomial of degree d.

Two properties of this scheme that are very useful for MPC are: (1) linear operations on secrets can be computed locally on the shares, since polynomial interpolation is a linear operation; (2) given shares of x and y, the parties can locally multiply their shares to obtain a sharing of degree $2d$ of $x \cdot y$.

In this work, we use a generalization of Shamir's sharing scheme where multiple secrets are being encoded together, introduced by Franklin and Yung [27] and known as "packed secret sharing". This is achieved by storing the secrets on multiple points. Note however that if we pack ℓ secrets

together on a polynomial of degree d, then the corruption threshold is being reduced to $t = d - \ell + 1$. Throughout this paper, we will use the notation $[\![x_1 \cdots x_\ell]\!]_d$ to denote a sharing of the block $x_1, \ldots, x_\ell$ using a polynomial of degree d, and assume that $x_1, \ldots, x_\ell$ are stored at points $0, -1, \ldots, -\ell + 1$ respectively and that the share of P_i is the value at the point i. Observe that the properties mentioned above apply to packed secert sharing as well. Namely, given a constant $\alpha, \beta \in \mathbb{F}$ and two sharings $[\![x_1 \cdots x_\ell]\!]_d$, $[\![y_1 \cdots y_\ell]\!]_d$, the following are local operations over the shares: (1) $[\![(\alpha x_1 + \beta y_1) \cdots (\alpha x_\ell + \beta y_\ell)]\!]_d = \alpha [\![x_1 \cdots x_\ell]\!]_d + \beta [\![y_1 \cdots y_\ell]\!]_d$; (2) $[\![x_1 y_1 \cdots x_\ell y_\ell]\!]_{2d} = [\![x_1 \cdots x_\ell]\!]_d \cdot [\![y_1 \cdots y_\ell]\!]_d$.

We say that a sharing $[\![x]\!]_d$ or $[\![x_1 \cdots x_\ell]\!]_d$ is inconsistent if all points do not lie on the same polynomial of degree d. Given all shares, this can be easily checked by using $d + 1$ points to reconstruct the polynomial and checking whether the remaining points lie on this polynomial

2.2 Computation Model: Layered Straight-Line Programs

In this work, we present a multi-party protocol for performing arithmetic computations over a finite field. In the MPC literature, arithmetic computations are usually represented by a circuit or a straight line program (SLP) with addition and multiplication gates/operations. We use the notion of SLP, but choose a slightly different representation, with one instruction, which we call "bi-linear", that captures the two operations together. This model will allow us a simple and more clearer description of our protocols, and in particular, make the trick to achieve "free-addition" easier to describe.

Definition 2.2 (Layered straight-line program (SLP)) *A* straight-line programs (SLP) *over $\mathbb{F}$ is defined by an arbitrary sequence of the following kinds of instructions:*

- *Load an input into memory: $R_j \leftarrow x_i$.*
- *Bilinear instruction: $R_j \leftarrow (\sum_{\omega=1}^{w} a_\omega \cdot R_\omega) \cdot (\sum_{\omega=1}^{w} b_\omega \cdot R_\omega)$*
- *Output value from memory, as element of $\mathbb{F}$: $O_i \leftarrow R_j$.*

Here $x_1 \ldots, x_n$ are inputs*, $R_1, \ldots, R_w$ are* registers *and $a_1, \ldots, a_w, b_1, \ldots, b_w$ are* public constants *in $\mathbb{F}$. We define the* size *of an SLP to be the number of instructions. A* layered SLP *is an SLP where the instructions are partitioned into sets called layers such that the inputs to instructions in layer j were computed in layer $k < j$. An ℓ-layered SLP is a layered SLP in which the number of instructions in each layer is a multiple of ℓ.*

For simplicity, we assume in our MPC protocols for SLP that each party holds one input and receives one output at the end. However, the protocols naturally extend to the general case of multiple inputs or outputs per party.

Remark 2.3 (Simulating arithmetic circuits by layered SLPs) *Every arithmetic circuit of size S (counting only multiplication gates, inputs, and outputs) can be converted into an SLP of size S by sorting its gates in an arbitrary topological order. The "ℓ-layered" notion of SLP intuitively corresponds to a lower bound on the circuit width. In particular, an SIMD circuit computing $k \geq \ell$ copies of a size-S circuit C on k distinct inputs can be written as an ℓ-layered SLP of size kS. Any layered SLP can be converted into an ℓ-layered one by naively adding dummy gates if needed, where the latter adds $(\ell - 1)$ times the depth in the worst case. But almost all "natural" instances of big circuits can be compiled into ℓ-layered SLPs with no overhead.*

2.3 MPC Security Definition

We use the standard definition of security based on the ideal/real model paradigm [15, 33] and augment it to capture an additional "stragglers-resilience" property. We consider in this work two types of adversaries. In Section 4, the adversary is semi-honest, which means that it follows the specification of the protocol but may try to learn private information. In Section 5, we augment the protocol to the more realistic model of malicious security, where the adversary can behave in an arbitrary way. For malicious security, when we say that a protocol securely computes an ideal functionality with abort, then we consider non-unanimous abort (sometimes referred to as "selective abort"). This means that the adversary first receives the output, and then determines for each honest party whether they will receive abort or their correct output. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution before the output is revealed [35], with constant communication cost; we therefore omit this step for simplicity.

3 Generalized Pseudorandom Secret Sharing

An important resource for our main protocol is a packed secret sharing of blocks of ℓ secrets that are randomly sampled from a given linear subspace. In this section, we show how the parties can securely generate arbitrarily many such blocks of secrets without any interaction, assuming a short setup step where they distribute seeds for a Pseudorandom Function (PRF). Subsequently, shares are obtained by local computation on the seeds. We refer to this problem as generalized pseudorandom secret sharing (PRSS). This primitive is useful beyond the context of this work, and our results are useful even without any share packing (i.e., when $\ell = 1$).

More abstractly, we can view the problem as that of efficiently realizing a *linear correlation*, namely an ideal functionality that picks a random vector from a public linear space and delivers one or more entries of this vector to each party. To be applicable in an MPC protocol, even with a semi-honest adversary, the linear correlations must be generated *securely*. Loosely speaking, an adversary should not get any information on the shares of honest parties beyond what follows from the public linear correlation, even given the information that the adversary holds.

The ideal functionality $\mathcal{F}_{\text{LinRand}}$. We will make security arguments relative to an ideal functionality $\mathcal{F}_{\text{LinRand}}$ for sharing instances of *linear correlated randomness*. More concretely, $\mathcal{F}_{\text{LinRand}}$ is parametrized by some linear subspace, and in each invocation it picks a random vector from that linear subspace and distributes one or more entries to each party. Both the linear space and the assignment of which entry goes to what party are public. It is only the actual vector sampled from the linear subspace that should remain secret.

Security is defined with respect to a *static* adversary who may corrupt up to t parties. Concretely, the real world view of the adversary together with the outputs of honest parties should be indistinguishable from an ideal world in which the adversary chooses the corrupted parties' shares, and then the honest parties' shares are sampled from the target correlation conditioned on this choice. This can be formally viewed as a multiparty instance of a Pseudorandom Correlation Function (PCF), recently defined by Boyle et al. [11], applied to *linear* correlations. The notion of PCF naturally extends the notion of a Pseudorandom Correlation Generators (PCG) [10], analogously to the way a standard PRF extends a standard PRG.

We are interested in t-secure realizations of $\mathcal{F}_{\text{LinRand}}$ that have the following structure: (1)

During an offline setup phase, a trusted dealer picks random and independent PRF seeds, and distributes each seed to a subset of the parties.[3] (2) Next, to realize a fresh invocation of $\mathcal{F}_{\text{LinRand}}$ with common identifier id, each party *locally* evaluates the PRF with each seed it owns on one or more inputs derived from id, and outputs a *fixed linear combination* of the PRF outputs. (The linear combination is fixed and does not change from one id to the next.)

3.1 Overview

Following prior work, we reduce the goal of secure realization of $\mathcal{F}_{\text{LinRand}}$ to an information-theoretic problem where the PRF seeds are replaced with true randomness. Namely, we consider locally generating an instance of the target correlation with perfect t-security given independently random field elements that are replicated between the parties. In the PRF-based computational realization of $\mathcal{F}_{\text{LinRand}}$, the random field elements will be pseudorandomly sampled using the PRF. Security under the above PCF-style definition reduces to information-theoretic security via a standard hybrid argument.

The PRSS problem was first implicitly studied by Gilboa and Ishai [32]. Cramer, Damgård, and Ishai [21] made this notion explicit and described a simple construction for the case of generating t-out-of-n Shamir sharing of random values. This construction is a useful building block in many cryptographic applications.

Here we extend the notion and construction of PRSS to more general settings that are motivated (among other applications) by MPC with strong honest majority. We show that a gap between the degree d and the security threshold t can give rise to dramatic efficiency improvements. Concretely:

- We start by extending the standard PRSS problem to the case where the degree of the Shamir-sharing polynomial can be larger than the security threshold t, and reduce this problem to a well-studied combinatorial design problem. This construction can be used for example to implement efficient distribution of *packed Shamir sharing* [27] of random values, and can be useful for many other applications.

- We show how to use the above construction in a black-box fashion to get efficient implementation of the kind of "double sharing" needed for protocols that follow the approach of Damgård-Nielsen (DN) [25]. Specifically, we implement the target correlation of two secret-sharing of the same (possibly packed) random values, one with a degree-d polynomial and the other with a degree-$2d$ polynomial.

- We show an extension of this technique to the harder case where we have degree-$2d$ sharings of random values, and degree-d sharings of *arbitrary linear combinations* of those random values. This is used to generate random packed secrets that satisfy given replicated constraints, as needed by efficient MPC protocols for *general circuits* based on packed secret sharing [22, 23].

We note that our techniques can be used to improve the efficiency of even more general forms of linear correlation, but leave systematic study of their application to future work.

3.2 The Gilboa-Ishai Framework

The functionality that we want to implement distributes linearly correlated random variables over some field $\mathbb{F}$ to n parties. The functionality is parameterized by a matrix $C \in \mathbb{F}^{N \times K}$ whose columns

[3] This setup can alternatively be implemented by a secure MPC protocol.

span a linear code (i.e., linear subspace of $\mathbb{F}^N$), and by a mapping $\rho : [N] \to [n]$ saying which party gets what entry of the output vector. The functionality chooses a random vector $\vec{v}$ in the code (by choosing a uniformly random $\vec{u} \leftarrow \mathbb{F}^K$ and setting $\vec{v} := C\vec{u}$), then privately sends to each party i' all the entries indexed by $\rho^{-1}(i')$.

Implementing this functionality without any interaction (beyond pre-distribution of PRF seeds) was studied by Gilboa and Ishai [32], in the information-theoretic setting where the PRF seeds are replaced by true randomness. In their framework, implementation of the linear-correlation functionality consists of:

- Input distribution, where an honest dealer draws $x_1, x_2, \ldots, x_k \in \mathbb{F}$ uniformly at random, and distributes each x_j to some subset of parties $S_j \subset [n]$;
- Local output computation, where each party i locally computes and outputs its entries of $\vec{v}$ from the x_j's that it received.

The complexity measures of interest for such a solution are:

- The number of distinct subsets S_j, corresponding to the number of PRF seeds to be distributed, and
- The sum $\sum_{j=1}^{k} |S_j|$, corresponding to the total number of pseudorandom field elements to be derived from these PRF seeds, across all the parties.

All the known implementations, including the ones that we describe here, rely on "small-support codewords" and the Gilboa-Ishai security criteria: A solution is specified by a "sparse" matrix $M \in \mathbb{F}^{N \times k}$ (typically $k \gg K$), whose columns span the same code as C. The output is computed by choosing a random vector $\vec{x} = (x_1, \ldots, x_k)$ and setting $\vec{v} := M\vec{x}$, and each party gets all the x_j'es that it needs in order to carry out this computation. Specifically, for an entry $\vec{v}[i]$ that belongs to party $\rho(i)$, we give that party the random elements x_j for which $M[i, j] \neq 0$, making it possible for this party to compute the inner product between $\vec{x}$ and the i'th row of M. Hence the sets $S_1, \ldots, S_k$ are defined

$$S_j = \{i' \in [n] : \exists i \in [N],\ M[i, j] \neq 0 \text{ and } i' = \rho(i)\}, \tag{1}$$

(For example, if the mapping ρ assigns entries 1 through 10 in $\vec{v}$ to Party 1 then the only x_j values that *are not given to this party* correspond to columns of M where the top 10 entries are all zero.) Clearly, the sparser the matrix M is, the fewer x_j values that must be distributed and the smaller we can make the sets S_j.

Gilboa and Ishai proved a necessary and sufficient criterion for security within this framework. Fix a code which is generated by the columns of the matrix C, and a solution matrix M whose columns span the same code. For a subset of parties $T \subset [n]$, let I_T be all the rows that belong to parties in T, and J_T be all the indices of x_j's that members of T get to see. That is, with the S_j's defined as in Equation (1), we have

$$I_T = \bigcup_{i' \in T} \rho^{-1}(i'), \text{ and } J_T = \big\{j \in [k] : S_j \cap T \neq \emptyset\big\}.$$

Denote by $\mathsf{C}_{\bar{T}}$ the restriction of $\mathsf{span}(C)$ to only the codewords that are zero in all the coordinates I_T. Also denote by $M'_{\bar{T}}$ the submatrix of M consisting of the rows in the complement $I_{\bar{T}} = [N] \setminus I_T$ and the columns in the complement $J_{\bar{T}} = [k] \setminus J_T$ (i.e., the ones corresponding to x_j's that *none of the parties in T receives*).

Lemma 3.1 ([32]) *Let $C \in \mathbb{F}^{N \times K}$ and $M \in \mathbb{F}^{N \times k}$ be two matrices with the same column space (so M describes a solution to the distribution of a codeword from $\mathsf{span}(C)$).*

For a subset of parties $T \subset [n]$, the solution specified by M is secure against a corrupted T iff the rank of $M'_{\bar{T}}$ equals the dimension of $\mathsf{C}_{\bar{T}}$.

3.3 Technical Tool: Covering Designs

The main technical tool that we use in our construction is the following notion of covering designs:

Definition 3.2 ((n,m,t)-cover) *Fix integers $n \geq m \geq t > 0$, and let $\mathcal{C} = (S_1, \ldots, S_k)$ be a collection of k different subsets $S_j \subset [n]$, all of size $|S_j| = m$. $\mathcal{C}$ is said to be an (n,m,t)-cover if for every size-t subset $T \subset [n], |T| = t$, there is a set $S_j \in \mathcal{C}$ that covers it, $T \subseteq S_j$. We will refer to an (n,m,t)-cover as a t-cover when n, m are clear from the context.*

This notion is equivalent to the notion of t-immunity of Alon et al. [2], in which for every subset T there is a set S_j in the collection such that $T \bigcap S_j = \emptyset$. The collection $\mathcal{C}$ is an (n,m,t)-cover iff the complement sets $[n] \setminus S_j$ form an $(n, n-m, t)$-immune collection. The smallest number of subsets in an (n,m,t)-cover is also known as the hypergraph Turán number $\mathcal{T}(n, n-t, n-m)$ in honor of Paul Turán who initiated the study of these objects in [57, 58].

The parameters of covering designs have been studied extensively, e.g. see [56, 28] for surveys, but the exact value is still an open problem in the general case. The best known analytical bounds for small values of t are summarized in a Handbook of Combinatorial Designs chapter by Gordon and Stinson [37]. A good online resource that collects the best known bounds for concrete values of n, m, t with $t \leq 8$, including ones found via computer search, is Gordon's covering designs web page [1].

For general values of t, Micali and Sidney [51] proposed to construct an (n,m,t)-cover by randomly choosing $\binom{n}{t}/\binom{m}{t} \ln \binom{n}{t}$ subsets of size m from $[n]$ and used a probabilistic argument to show that with good probability this collection is an (n,m,t)-cover. Pieprzyk and Wang [44] construct a deterministic, greedy algorithm that achieves the same bound on the size of the collection. Both works were motivated by variants of the PRSS problem where the seeds are stored in a replicated form, without the compressing share conversion step from [32, 21]

A range of parameters which is especially appealing for our MPC protocol is constant t, and m which is a linear fraction of n, e.g., $m = n/3$. In this case, the protocol can cope with a large number of stragglers and reduce communication by packing. When t is constant, the constructions in [51, 44] have collections of size $O(\log n)$.

We next describe a simple construction that achieves a constant-sized collection for $t = O(1)$ and $m = \Omega(n)$, when t divides m and m divides n. Denote $c = n/m$ and partition $[n]$ into ct subsets $R_1, \ldots, R_{ct}$ of equal size. Let the collection $S_1, \ldots, S_k$ be all the possible choices of t subsets $R_{i_1} \cup \cdots \cup R_{i_t}$. Obviously, each $|S_j| = t(n/ct) = m$ and for every $T \subseteq [n], |T| = t$ there exists some S_j such that $T \subseteq S_j$. The size of the collection is $\binom{ct}{t}$, which for constant t and c is constant, improving over the construction of [51, 44].

Taking for each parameter set (n,m,t) the minimal cover size between the simple construction and the construction in [44] provides a baseline construction for t-covers. This baseline achieves an upper bound for the cover size, which is bigger than the minimal possible size by a factor of at most $O(\log n)$, due to a simple lower bound of $\binom{n}{t}/\binom{m}{t}$ on this size (see, e.g., Theorem 11.19 in [37]). Both the upper bound of the baseline construction and the simple lower bound are generally not tight. Improved bounds for certain parameter ranges can be found in [1].

(n, m, t)	Baseline cover size	Best known cover size	Lower bound cover size	CDI seeds per party	PRSS seeds per party
$(9, 3, 1)$	3	3	3	8	7
$(15, 5, 1)$	3	3	3	14	11
$(15, 5, 2)$	49	13	13	91	48
$(48, 16, 1)$	3	3	3	47	33
$(48, 16, 2)$	15	13	13	1081	143
$(48, 16, 4)$	495	252	173	178365	2772
$(48, 20, 4)$	490	87	60	178365	1052
$(48, 20, 6)$	5168	1280	459	$1.07 \cdot 10^6$	15467
$(49, 24, 2)$	31	7	7	1128	90
$(49, 24, 4)$	245	38	31	194580	484
$(49, 24, 8)$	12219	4498	968	$3.7 \cdot 10^8$	57281
$(72, 24, 2)$	15	12	12	2485	196
$(72, 24, 4)$	495	180	126	971635	2940
$(72, 24, 6)$	18564	4998	1419	$1.4 \cdot 10^8$	81634

Table 1: Concrete bounds for (n, m, t)-covers and generalized PRSS. The baseline cover size captures a simple upper bound given by the minimum between $\binom{n}{t}/\binom{m}{t} \ln \binom{n}{t}$ and $\binom{nt/m}{t}$ when applicable, i.e. when m/t and n/m are integers. The best known cover size and the lower bound on the cover size are given by [1]. The "CDI seeds per party" column refers to the variant of the construction from [21] described in Footnote 2, which requires $\binom{n-1}{t}$ seeds per party. The "PRSS seeds per party" column refers to the PRSS given by Theorem 3.3 for the linear correlation defined by random polynomials of degree $m-1$ using the best known cover size k, namely multiplying k by $m(n-m+1)/n$.

Table 1 includes upper and lower bounds on the cover size for useful parameters (n, m, t). In addition, it shows the number of seeds for PRSS used to distribute random degree d polynomials, for $d = m - 1$, using the construction we present next.

3.4 Generalized PRSS for Degree-d Polynomials

It is easy to see (see Theorem 3.5) that t-covers are necessary for t-secure distribution in the Gilboa-Ishai framework, since any corrupted subset must miss at least some of the x_j's. Here we observe that the other direction is also useful, establishing a close connection between the size of the best $(n, d+1, t)$-cover and the complexity of PRSS for distributing random degree-d polynomials between n parties with security against t-collusions.

Theorem 3.3 (Generalized PRSS for degree-d polynomials) *Fix integers $n \geq d > t > 0$. A size-k' $(n, d+1, t)$-cover can be used to construct a generalized PRSS solution for t-secure distribution of degree-d polynomials, with the following complexity measures:*

- *The number of distinct subsets (or PRSS seeds) is $k = k'(d+1)$, and*
- *The total subset size (storage) is $\sum_i |S_i| = k'(d+1)(n-d)$ and*
- *The total number of PRF calls is $k'(d+1)(n-d)$.*

Proof: Let $\mathcal{C}' = (S'_1, \ldots, S'_{k'})$ be a size-k' $(n, d+1, t)$-cover, i.e. it consists of k' subsets, each of size $d+1$, that cover all the t-subsets. We then consider all the subsets that are obtained by removing one element from any of the S'_j's,

$$\bar{\mathcal{C}} = \left\{ S' \setminus \{j\} : \ S' \in \mathcal{C}', j \in S' \right\}.$$

Clearly, there are at most $k \leq k'(d+1)$ distinct subsets in $\bar{\mathcal{C}}$, each of size d. Let us denote the subsets in $\bar{\mathcal{C}}$ by $\bar{S}_1, \bar{S}_2, \ldots, \bar{S}_k$, and we use these subsets in the CDI construction to distribute a random degree-d polynomial. We let $P_{\bar{S}_j}$ be the unique polynomial of degree d interpolated from

$$P_{\bar{S}_j}(X) = \begin{cases} 0 & \text{if } X \in \bar{S}_j \\ 1 & \text{if } X = 0 \end{cases}$$

As before, $P_{\bar{S}}$ is a nonzero degree-d polynomial, whose zeros are exactly all the parties in $\bar{S}_j$. A random vector $\vec{x} \in \mathbb{F}^k$ therefore defines the polynomial $Q_{\vec{x}}(X) = \sum_j x_j \cdot P_{\bar{S}_j}(X)$, and each party $i \in [n]$ gets the x_j's corresponding to the $\bar{S}_j$'s *that do not include* i, and can compute $Q_{\vec{x}}(i)$ from these x_j's. Thus, there are $k'(d+1)$ distinct subsets, each of cardinality $n-d$. This implies that the total stroage is $k'(d+1)(n-d)$ as the theorem states. Since each seed is used once, the total number of PRF calls is also the same.

In the language of the Gilboa-Ishai framework, the matrix $M \in \mathbb{F}^{n \times k}$ is defined by $M[i,j] = P_{\bar{S}_j}(i)$, and the distribution sets are exactly the complementing sets $S_j = [n] \setminus \bar{S}_j$ (namely we distribute each x_j to the complement of some $S' \in \mathcal{C}'$, together with one more element). The complexity measures are obvious.

It remains therefore to show security against a collusion of t parties, which for degree-d polynomials means showing that for every t-subset T, the submatrix $M'_{\bar{T}}$ has rank at least $d+1-t$. Fix a t-subset $T \subset [n]$, so there is a subset $S' \in \mathcal{C}'$ that covers it. Consider now the sub-matrix corresponding to the subsets $\bar{S}$ that were obtained by removing from S' one element *which is not in* T (hence those sets $\bar{S}$ still all cover T). That is, we consider the sub-matrix $M_{T,S'}$ of $M[i,j] = P_{\bar{S}_j}(i)$, consisting of the rows for $[n] \setminus T$ and the columns for $S_j = ([n] \setminus S') \cup \{j'\}$ for all $j' \in S' \setminus T$. Clearly $M_{T,S'}$ is a sub-matrix of $M'_{\bar{T}}$, it has $n-t$ rows and $d+1-t$ columns (since S' covers T), and it has the form

$$M_{T,S'} = \begin{bmatrix} * & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ * & * & \cdots & * \\ * & & & \\ & & \ddots & \\ & & & * \end{bmatrix},$$

where the $*$'s are non-zero and everywhere else there are zeros. The top rows $* \cdots *$ correspond to $[n] \setminus S'$ and the bottom rows correspond to $S' \setminus T$. The last $d+1-t$ rows of this matrix are linearly independent, hence the rank of $M_{T,S'}$ is $d+1-t$, as needed for the Gilboa-Ishai condition. ■

Corollary 3.4 *Fix integers $n \geq d > 1$. Then, the following holds for generalized PRSS solutions for t-secure distribution of degree-d polynomials with $t = 1, 2$:*

1. *There exists a solution for* $t = 1$ *with* $\left\lceil \frac{n}{d+1} \right\rceil (d+1)$ *total seeds,* $\left\lceil \frac{n}{d+1} \right\rceil \frac{(d+1)(n-d)}{n}$ *seeds stored by each party and* $\left\lceil \frac{n}{d+1} \right\rceil \frac{(d+1)(n-d)}{n}$ *calls to the PRF made by each party.*

2. *If* $n \leq 3(d+1)$ *then there exists a solution for* $t = 2$ *with* $13(d+1)$ *total seeds,* $13(d+1)(n-d)/n$ *seeds stored by each party and* $13(d+1)(n-d)/n$ *calls to the PRF made by each party.*

Proof: The corollary follows directly from Theorem 3.3 by plugging the best covering designs for $t = 1, 2$. If $t = 1$ then the best covering design is achieved by arbitrarily dividing the n parties into enough sets of size $d+1$ to cover all the parties which leads to a cover of size $k' = \left\lceil \frac{n}{d+1} \right\rceil$.

If $t = 2$ and $n \leq 3(d+1)$ then the best design is of size at most 13 [37]. ■

We can also prove a nearly-matching lower bound Theorem 3.3 on the solution complexity for t-secure distribution of degree-d polynomials, in terms of the achievable size for $(n, d+1, t)$-covers. This naturally generalizes a similar negative result for standard PRSS from [21].

Theorem 3.5 (Necessity of cover designs) *Any generalized PRSS solution for* t*-secure distribution of degree-*d *polynomials that has* k *distinct subsets implies an* $(n, d+1, t)$*-cover of size* $k' \leq k$*.*

Proof: Let M be a solution and $\mathcal{C} = (S_1, \ldots, S_k)$ be the distinct subsets used by that solution. By definition, each column of M is a non-zero codeword, namely the evaluation of a non-zero degree-$\leq d$ polynomial at the points $[n]$, so at most d of them are zero and the corresponding set S_j has size at least $n - d$.

Fix one of the parties (say Party 1). Consider all the subsets that include that party, but remove that party from all of them. Namely, set $\mathcal{C}' = \{S_i \setminus \{1\} : 1 \in S_i\}$. $\mathcal{C}'$ is a collection of $k' \leq k$ subsets, each of size at least $n - d - 1$. Let $\bar{\mathcal{C}} = \{[n] \setminus S' : S' \in \mathcal{C}'\} = \{\bar{S}_1, \ldots, \bar{S}_{k'}\}$ be the collection of complementing subsets, so it is a collection of k' subsets of size at most $d+1$. We now argue that every t-subset of $[n]$ must be contained by one of these $\bar{S}_i$'s.

Consider first a subset $T \subset [n], |T| = t$ that *does not include* Party 1. If T intersects every subset $S' \in \mathcal{C}'$ then Party 1 has no random value x_j that is not known to the parties in T, hence the evaluation $Q(1)$ is known to a corrupted subset T, contradicting security. For a subset T that includes Party 1, we note that by construction $1 \notin S'_i$ for all i. Hence, if T intersects every subset $S' \in \mathcal{C}'$ then $T' = T \setminus \{1\}$ intersects every subset $S' \in \mathcal{C}'$. The same argument from above implies that the evaluation point $Q(1)$ is known to the corrupted parties in T'. ■

The combination of Theorems 3.3 and 3.5 prove that the best $(n, d+1, t)$-cover implies a nearly optimal number of distinct subsets, up to a factor of at most $d+1$.

3.5 Double Shamir Sharing

A useful resource for efficient honest-majority MPC protocols is a so-called "double Shamir sharing" of a random secret, where the parties are given two random polynomials of degrees d and $2d$ that share the same random secrets. Here we consider the case of packed secret sharing. Letting $\ell = d - t + 1$ be the packing parameter, we want to generate a random degree-d polynomial P_1, and another polynomial P_2 of degree-$2d$ which is random subject to $P_1(x) = P_2(x)$ for all $x \in \{0, -1, -2, \ldots, -\ell + 1\}$. It is easy to see that this task reduces to generating two independent random polynomials $P_1(X)$ of degree d and $R(X)$ of degree $2d - \ell$, then setting

$$P_2(X) = P_1(X) + R(X) \cdot X(X+1)(X+2)\cdots(X+\ell-1).$$

Indeed, the polynomial on the right side is a random degree-$2d$ polynomial, under the constraint that its values at the points $\{0, -1, \ldots, \ell+1\}$ are 0. Since $P_1(x)$ and $R(x)$ are random independent polynomials, we can use the construction from the previous section in a black-box way. Specifically, we can generate $P_1(x)$ using a $(n, d+1, t)$-cover and generate $R(x)$ using an $(n, 2d-\ell+1, t)$-cover.

Theorem 3.6 (Generalized PRSS for packed double sharing) *Fix integers $d > t > 0$ and $n > 2d$ and let $\ell = d - t + 1$. A size-k' $(n, d+1, t)$-cover and a size-k'' $(n, 2d-\ell+1, t)$-cover can be used to construct a solution for t-secure distribution of double-Sharing of degree-d and degree-$2d$ polynomials, both packing the same ℓ elements, with the following complexity measures:*

- *The number of distinct subsets (seeds) is at most $k \leq k'(d+1)+k''(2d-\ell+1) \leq k'(2d+t+1)$;*
- *The total subset size (storage) is $\sum_j |S_j| \leq k'(d+1)(n-d) + k''(d+t)(n-d-t+1)$.*
- *The total number of PRF calls is $k'(d+1)(n-d) + k''(d+t)(n-d-t+1)$.*

Proof: The complexity measures follow from Theorem 3.3 and using the fact P_1 is a random polynomial of degree d and R is a random polynomial of degree $2d - \ell = d + t + 1$. Specifically, by Theorem 3.5 we have: (i) the number of distinct subsets is $k'(d+1) + k''(2d-\ell+1) \leq k'((d+1)+(d+t)) = k'(2d+t+1)$; (ii) the subsets generated from the first cover are of cardinality $n-d$, whereas the subsets generated from the second cover are of cardinality $n-(2d-\ell) = n-d-t+1$, and so the number of seeds that are overall distributed is $k'(d+1)(n-d)+k''(d+t)(n-d-t+1)$; (iii) since each seed is used once by each party that holds it, we have that the total number of calls to a PRF is $k'(d+1)(n-d) + k''(d+t)(n-d-t+1)$ as well. Security follows since both P_1 and R are generated with t-security and their sum is also t-secure. ■

This construction is already strong enough to support DN-type secure computation protocols, even while packing ℓ elements in each polynomial. (Hence it can be used to compute the same circuit on ℓ different inputs at once, in a SIMD fashion.)

As an alternative to the above, we can use an $(n, d+1, t)$-cover to construct both polynomials, by increasing the number of pseudorandom elements derived from each seed. This will reduce the number of seeds stored by the parties (by some factor smaller than two), but will increase the number of pseudorandom elements that must be derived from these seeds. For completeness, we provide the construction in Appendix B. We use a similar idea in the construction in the next section.

3.6 Beyond Double Sharing

In some applications, including the protocol that we describe in Section 4, we must generate double-Shamir-sharing of linearly correlated packed values (rather than the same values twice). While we don't know how to use the random-polynomial construction in a black-box manner to achieve this, we show here how to modify that construction in order to distribute this more general linear correlation in a t-secure manner.

This extension, however, comes with some loss of efficiency. Specifically, we need to start from covers with smaller subsets, and moreover we no longer distribute only a single random element to each subset. Fix $n > d > t > 0$ and $\ell \leq d - t$ (allowing $\ell < d - t$ is useful to mitigate the parameter loss). The goal in this section is to share two types of polynomials:

- m polynomials $R_1, \ldots, R_m$ of degree $2d$, each packing ℓ "free variables" (i.e. unconstrained) in positions $0, -1, \ldots, -\ell+1$.
- m' additional polynomials $U_1, \ldots, U_{m'}$ of degree d, each packing ℓ constrained variables, which are set as some fixed linear combinations of the free variables.

Denote the positions where these values are packed by $L = \{0, -1, \ldots, -\ell+1\}$, and also denote the linear correlation above by $\mathcal{L}[n, d, \ell, m, m']$.

Let us give the high-level ideas of the construction. The polynomials R_α are generated as the sum of polynomials $R_{\alpha,\gamma}$ that pack a random value at position γ and 0 in all the other positions (i.e., $R_{\alpha,\gamma}(\gamma)$ is random and $R_{\alpha,\gamma}(X) = 0$ if $X \in L \setminus \{\gamma\}$). The polynomials $R_{\alpha,\gamma}$ are generated similarly to Section 3.4 with the following difference: the underlying polynomials $P_{1,\bar{S}_j,\gamma}(X)$ are constrained to be 1 at positions γ and 0 in all other positions (as opposed to being 1 at position 0 and unconstrained at the other positions $-1, \ldots, \ell+1$ in Section 3.4). Then, the i-th shares $U_\beta(i)$ of the correct polynomials $U_\beta(X)$ can be computed by the values used by party i to generate the polynomials R_α. These shares need to be re-randomized by adding random polynomials of degree d that pack 0 at each position $-1, \ldots, \ell+1$. These last polynomials are generated similarly to Section 3.4 with the following difference: the underlying polynomials $P_{2,\bar{S}_j}(X)$ are constrained to be 0 in all positions $0, \ldots, -\ell+1$ and 1 at position $-\ell$.

Theorem 3.7 (Generalized PRSS for replicated packed secrets) *Fix integers $n \geq d > t > 0$, $\ell \leq d - t$, $m, m' > 1$. A size-k' $(n, d-\ell+1, t)$-cover can be used to construct a solution for t-secure distribution of the linear correlation $\mathcal{L}[n, d, \ell, m, m']$ above. The complexity is at most:*

- *The number of distinct subsets (seeds) is at most $k \leq k'(d-\ell+1)$;*
- *The total subset size (storage) is $\sum_j |S_j| \leq k'(d-\ell+1)(n-d+\ell)$;*
- *The total number of PRF calls is at most $k(n-d+\ell)(m(d+\ell+1)+m')$.*

Proof: Let the relation $\mathcal{L}[n, d, \ell, m, m']$ be defined by the coefficients $\lambda_{\beta,z,\alpha,y}$ where $\alpha \in [m]$, $\beta \in [m']$, and $y, z \in L$, as follows:

$$U_\beta(z) = \sum_{\alpha \in [m]} \sum_{y \in L} \lambda_{\beta,z,\alpha,y} R_\alpha(y) \ .$$

We will use a $(n, d-\ell+1, t)$-cover $\mathcal{C} = \{S'_1, S'_2, \ldots, S'_{k'}\}$. As in section 3.4 we remove all possible singletons from each S'_j to get the collection

$$\bar{\mathcal{C}} = \{\bar{S}_1, \ldots, \bar{S}_k\} = \left\{S' \setminus \{j\} : \ S' \in \mathcal{C}', j \in S'\right\}.$$

As before, we have $k \leq k'(d-\ell+1)$ subsets, each of size $d-\ell$.

We first use the sets $\bar{S}_j$ to generate random degree-$2d$ polynomials, packing the unconstrained variables. For every degree-$2d$ polynomial R_α ($\alpha \in [m]$) that we want to share, for every $\gamma \in \{0, \ldots, -d-\ell\}$, and for every subset $\bar{S}_j$ ($j \in [k]$), the dealer distributes a random element $x_{1,\alpha,\gamma,j}$ to parties in the complement sets $S_j = [n] \setminus \bar{S}_j$. Next let $P_{1,\bar{S}_j,\gamma}$ be the unique polynomial of degree $2d$ interpolated from

$$P_{1,\bar{S}_j,\gamma}(X) = \begin{cases} 0 & \text{if } X \in \bar{S}_j \\ 0 & \text{if } X \in \{0, \ldots, -d-\ell\} \setminus \{\gamma\} \\ 1 & \text{if } X = \gamma \end{cases},$$

and set $R_{\alpha,\gamma,j}(X) = x_{1,\alpha,\gamma,j} \cdot P_{1,\bar{S}_j,\gamma}(X)$ for all $\alpha \in [m]$ and $j \in [k_1]$. This polynomial has degree $2d$ as needed, and it is known fully to every party $i \in S_j$. We finally set $R_\alpha(X) = \sum_{j,\gamma} R_{\alpha,\gamma,j}(X)$, for $\alpha \in [m]$. Note that the share $R_\alpha(i) = \sum_{j \text{ s.t. } i \in S_j} x_{1,\alpha,\gamma,j} \cdot P_{1,\bar{S}_j,\gamma}(i)$ of party i can be computed just from the values $x_{1,\alpha,\gamma,j}$ known by party i (i.e., such that $i \in S_j$). Note also that in the language of the Gilboa-Ishai framework, each set of parties S_j would get $m(d+\ell+1)$ of the $x_{1,\alpha,\gamma,j}$ (so this set will appear $m(d+\ell+1)$ times in the list).

We now proceed to select the polynomials U_β. For every degree-d polynomial U_β ($\beta \in [m']$) that we want to share, and for every subset $\bar{S}_j$ ($j \in [k]$), the dealer distributes a random element $x_{2,\alpha,j}$ to parties in the complement sets $S_j = [n] \setminus \bar{S}_j$. For $y, z \in L$ and $\gamma \in \{0, \ldots, -d-\ell\}$, let $Q_{1,\bar{S}_j,y,z,\gamma}$ be the unique polynomial of degree $\leq d$ interpolated from:

$$Q_{1,\bar{S}_j,y,z,\gamma}(X) = \begin{cases} 0 & \text{if } X \in \bar{S}_j \\ 0 & \text{if } X \in L \setminus \{z\} \\ P_{1,\bar{S}_j,\gamma}(y) & \text{if } X = z \\ 1 & \text{if } X = -\ell \end{cases},$$

and let $P_{2,\bar{S}_j^2}$ be the unique polynomial of degree d interpolated from

$$P_{2,\bar{S}_j}(X) = \begin{cases} 0 & \text{if } X \in \bar{S}_j \\ 0 & \text{if } X \in L \\ 1 & \text{if } X = -\ell \end{cases},$$

and set

$$U_{\beta,j}(X) = \sum_{\alpha \in [m]} \sum_{y,z \in L} \sum_{\gamma} \lambda_{\beta,z,\alpha,y} x_{1,\alpha,\gamma,j} Q_{1,\bar{S}_j,y,z,\gamma}(X) \; + \; x_{2,\beta,j} P_{2,\bar{S}_j}(X) \;\; ,$$

(the first term of the sum is used to ensure that the linear combinations are satisfied, while the second term is used to re-randomize the polynomial U_β without breaking the correctness) and $U_\beta(X) = \sum_{j \in [k]} U_{\beta,j}(X)$.

Since $U_{\beta,j}(i) = 0$ when $i \notin S_j$, $U_\beta(i)$ can be computed just from the values $x_{1,\alpha,\gamma,j}$ and $x_{2,\beta,j}$ known by party i.

Let us now prove that $U_\beta(X)$ actually pack the correct linear combinations of the free variables. We remark that:

$$Q_{1,\bar{S},j,y,z,\gamma}(z') = \begin{cases} P_{1,\bar{S},j,\gamma}(y) & \text{if } z = z' \\ 0 & \text{if } z \in L \setminus \{z'\} \end{cases}$$
$$P_{2,\bar{S}_j}(z) = 0 \;\; .$$

Thus we have:

$$\begin{aligned} U_\beta(z) &= \sum_{\alpha \in [m]} \sum_{y} \lambda_{\beta,z,\alpha,y} \cdot \sum_{z' \in L} \sum_{j,\gamma} x_{1,\alpha,\gamma,j} Q_{1,\bar{S}_j,y,z,\gamma}(z') + \sum_j x_{2,\alpha,j} P_{2,\bar{S}_j}(z) \\ &= \sum_{\alpha \in [m]} \sum_{y} \lambda_{\beta,z,\alpha,y} \cdot \sum_{j,\gamma} x_{1,\alpha,\gamma,j} P_{1,\bar{S}_j,\gamma}(z) \\ &= \sum_{\alpha \in [m]} \sum_{y,z' \in L} \lambda_{\beta,z,\alpha,y} \cdot R_\alpha(z) \;\; . \end{aligned}$$

This is what is required for correctness.

Finally, let us prove the security of construction. Fix a t-subset $T \subset [n]$. We need to prove that the submatrix $M'_{\bar{T}}$ (as defined in theorem 3.1) has rank at least $(2d+1-t)m + (d+1-\ell-t)m'$. There is a subet $S' \in \mathcal{C}'$ that covers T. Let $\kappa = d-\ell+1-t$. Write $S' \setminus T = \{i_1, \ldots, i_\kappa\}$ and let j_δ be such that $\bar{S}_{j_\delta} = S' \setminus \{i_\delta\}$.

We remark that we can write:

$$R_\alpha(i_\delta) = \sum_\gamma x_{1,\alpha,\gamma,j_\delta} P_{1,\bar{S}_{j_\delta},\gamma}(i_\delta) + r'_{\alpha,i_\delta} = \sum_\gamma x_{1,\bar{S}_{j_\delta},\gamma} + r'_{\alpha,i_\delta}$$

where r'_{α,i_δ} does not contain any $x_{1,\alpha,\gamma,j_{\delta'}}$. Furthermore:

$$R_\alpha(\gamma) = \sum_{\delta'} x_{1,\bar{S}_{j_{\delta'}},\gamma} + r'_{\alpha,i_\delta} \ ,$$

where r'_{α,i_δ} does not contain any $x_{1,\alpha,\gamma,j_{\delta'}}$. We can multiply $M'_{\bar{T}}$ to the left by a matrix without loss of generality, as this can only decrease the rank. Doing so, we can make the rows corresponding to R_α to correspond to evaluation points $\{i_1, \ldots, i_\kappa, 0, -1, \ldots, -d-\ell\}$ (there are $2d+2-t$ such points) instead of evaluation original evaluation points $\{1, \ldots, n\}$. Looking at columns corresponding to $x_{1,\bar{S}_{j_{\delta'}},\gamma'}$, the resulting matrix looks like:

$$\begin{pmatrix}
1 & & & 1 & & & \cdots & & 1 & & \\
& \ddots & & & \ddots & & & & & \ddots & \\
& & 1 & & & 1 & \cdots & & & & 1 \\
1 & \cdots & 1 & & & & & & & & \\
& & & 1 & \cdots & 1 & & & & & \\
& & & & & & \ddots & & & & \\
& & & & & & & & 1 & \cdots & 1
\end{pmatrix},$$

where the top part has $d+\ell+1$ rows and the bottom part has κ rows. Subtracting the first column to each of the column with a one in the first row, and eliminaing all the columns with a non-zero coefficient in the top part (except the first $d+\ell+1$ columns), we get the following $(d+\ell+1+\kappa) \times (d+\ell+1+\kappa-1)$-matrix:

$$M'_\alpha = \begin{pmatrix}
1 & & & & & & \\
& \ddots & & & & & \\
& & 1 & & & & \\
1 & \cdots & 1 & -1 & \ldots & -1 \\
& & & 1 & & & \\
& & & & \ddots & & \\
& & & & & 1
\end{pmatrix}$$

This matrix has rank $d+\ell+1+\kappa-1 = 2d+1-t$.

Finally, up to multiplication by a matrix to the left and removing some columns, the matrix

$M'_{\bar{T}}$ can be written in the following form:

$$\begin{pmatrix} M'_1 & & & & & \\ & \ddots & & & & \\ & & M'_m & & & \\ * & \dots & * & M''_1 & & \\ \vdots & & \vdots & & \ddots & \\ * & \dots & * & & & M''_{m'} \end{pmatrix}$$

where M''_β is the $\kappa \times \kappa$-identity matrix and corresponds to some of the columns $x_{2,\beta,j_{\delta'}}$. The rank is thus $(2d+1-t)m + \kappa m'$. This concludes the proof. ■

Parameters. We remark that the parameters of this construction behave differently than those of the previous constructions. For the constructions from Sections 3.4 and 3.5, increasing ℓ (and d) was a double-win, not so for the current construction. Here we need to start from a $(n, d-\ell+1, t)$-cover, so setting $\ell = d-t$ we hardly get any slackness in the size of the sets in our t-cover (they will be of size only $t+1$). To improve parameters (the cover size in particular), it is better to choose a smaller value of ℓ, thereby working with larger subsets and hence being able to find smaller covers. It is likely that setting $\ell \approx (d-t)/2$ will be a sweet spot for this construction in terms of complexity.

Remark 3.8 (Size of the field $\mathbb{F}$) *Note that polynomial $P_{1,\bar{S}_j,\gamma}(X)$ defined in the construction of Theorem 3.7 requires $n+d+\ell+1$ evaluation points. Thus, throughout the paper, we require that $|\mathbb{F}| \geq n+d+\ell+1$.*

4 Constructions for Semi-Honest Security

In this section, we present protocols to compute a layered straight-line program over a finite field $\mathbb{F}$, that is secure in the presence of a *semi-honest* adversary who controls t parties, and with straggler-resilience. Recall that we have $n \geq 2d+1$ parties, where $d \geq t+\ell-1$.

The starting point of our constructions is the DN protocol [25], which is the fastest protocol known to this date for $n > 3$ parties. We begin in Section 4.1 with recalling the baseline DN protocol. In Section 4.2, we introduce straggler resilience and show how to adapt the DN protocol accordingly. Then in Section 4.3 we provide our solutions for improving the communication and computation requirements of the protocol.

4.1 Baseline Protocol (with $\ell = 1$)

Recall that in the DN protocol [25], the parties compute linear operations without any interaction and compute multiplication operations with small constant communication cost per party. Given shares $[\![x]\!]_d$ and $[\![y]\!]_d$, the parties compute $[\![x \cdot y]\!]_d$ in the following way. The parties prepare random sharings $[\![r]\!]_d$ and $[\![r]\!]_{2d}$ in an offline step which are consumed as follows. First, the parties locally compute $[\![x \cdot y - r]\!]_{2d} = [\![x]\!]_d \cdot [\![y]\!]_d - [\![r]\!]_{2d}$ and send their shares to P_1. Then, party P_1 computes $x \cdot y - r$ and shares the result to the parties as $[\![xy-r]\!]_d$. Finally, the parties locally compute $[\![x \cdot y]\!]_d = [\![r]\!]_d + [\![xy-r]\!]_d$.

As the random sharings can be generated non-interactively (in the way described in Section 3), the communication cost is derived from parties sending one field element to P_1 and P_1 secret sharing $xy-r$ to the parties. Note that $2d$ shares are sufficient for P_1 to reconstruct $xy-r$ (together with its own share). Also, it is possible to reduce communication in the second round by setting the shares of d parties to be 0, and having P_1 define its own share and the remaining $n-d$ parties' shares, given the value of $xy-r$ and the d zero shares. This is possible since $xy-r$ is not secret (P_1 could send it in the clear to the parties) and since $[\![xy-r]\!]_d$ is shared via a polynomial of degree d, and so $d+1$ points are sufficient to define it. Overall, we have that the communication cost per party per bilinear gate is $\frac{2d+n-d-1}{n} = 1+\frac{d-1}{n}$ field elements. When $n > 2d+1$, it is possible to improve this by having the parties secret sharing their inputs to $2d+1$ parties who perform the computation. In this case, the communication cost per party per bilinear gate reduces to $\frac{2d+d}{n} = \frac{3d}{n}$ elements.

We denote by $\Pi_{\mathrm{SH}}^{\mathsf{base}}$ the base protocol, which thus works as follows:

Protocol $\Pi_{\mathrm{SH}}^{\mathsf{base}}$:

The parties hold a description of a layered SLP over $\mathbb{F}$. Denote by S the set of parties $P_1, \ldots, P_{2d+1}$

- **Pre-processing**: The parties call $\mathcal{F}_{\mathrm{LinRand}}$ to obtain a pair of random sharings $[\![r]\!]_d$ and $[\![r]\!]_{2d}$ for each bilinear instruction.
- **The protocol**:
  1. *Input sharing:* for each instruction $R_j \leftarrow x_i$, party P_i run $[\![x_i]\!]_d \leftarrow \mathsf{share}(x_i)$ and sends the resulting shares to the parties in S.
  2. *Evaluating the jth bilinear instruction* $R_j \leftarrow (\sum \alpha_\omega R_\omega) \cdot (\sum \beta_\omega R_\omega)$: Let $[\![r]\!]_d, [\![r]\!]_{2d}$ be the next unused pair of random sharings. Then:
     (a) The parties in S locally compute $[\![x]\!]_d = \sum_{\omega=1}^{w} \alpha_\omega \cdot [\![R_\omega]\!]_d$ and $[\![y]\!]_d = \sum_{\omega=1}^{w} b_\omega \cdot [\![R_\omega]\!]_d$, where $[\![R_\omega]\!]_d$ denotes sharing of the ω-index memory value R_ω (stored from previous operations).
     (b) The parties in S locally compute $[\![xy-r]\!]_{2d} = [\![x]\!]_d \cdot [\![y]\!]_d - [\![r]\!]_{2d}$ and send the result to P_1.
     (c) P_1 locally reconstructs $xy-r$ and then computes a sharing $[\![xy-r]\!]_d$ such that the shares of $P_2 \ldots, P_{d+1}$ are 0. Then, it sends the non-zero shares to parties $P_{d+2}, \ldots, P_{2d+1}$.
     (d) The parties in S set $[\![z]\!]_d \leftarrow [\![r]\!]_d + [\![xy-r]\!]_d$, and define $[\![z]\!]_d$ as their share of the output.
  3. *Output reconstruction:* For each instruction $O_i \leftarrow R_j$, the parties in S send their shares of the value in R_j to P_i, who uses them to reconstruct the output O_i.

Security of $\Pi_{\mathrm{SH}}^{\mathsf{base}}$ against a semi-honest adversary $\mathcal{A}$ controlling d parties follows from the fact that $\mathcal{A}$'s view consists of d random shares in the input sharing step, and masked intermediate values when performing multiplication operations.

4.2 Straggler Resilience

The classical communication model for secure multi-party computation considers parties who advance in the same pace in a fully synchronous manner. However, in real world scenarios, it is

unreasonable to assume that all messages arrive at the same time. A protocol which can proceed without having to wait for all the parties' messages to arrive in each round, has thus the potential to reduce the overall latency of the execution.

We consider a model of *straggler resilience*, to account for the fact that communication channels exhibit a distribution over latency times, each of which may incur long delays with small probability. Instead of requiring parties to block and wait in every communication round until the last messages arrive, we build into the protocol design that the computation may proceed even in the absence of a small number of messages per round, which have not yet successfully been delivered. We say that a protocol that terminates successfully even when τ messages are dropped in each round, is resilient to τ stragglers. As for privacy, following the standard definition of multi-party computation [33], we consider an adversary who controls t parties and, in addition, is allowed to choose τ messages to be dropped in each round.

Definition 4.1 (Straggler resilience, semi-honest security) *Let f be an n-party functionality. We say that protocol Π computes f with t-semi-honest-security and τ-straggler-resilience if it satisfies the following properties:*

- Straggler-Robust Correctness: *Π terminates successfully (i.e. each party receives its prescribed output $f_i(\vec{x})$), even if in each communication round, τ messages, chosen adaptively by the adversary, are not delivered.*
- Semi-Honest Security with Stragglers: *For every real-world semi-honest adversary $\mathcal{A}$ controlling a set I of parties with $|I| \leq t$ and, in addition, can choose adaptively τ messages to drop in each communication round, there exists an ideal-world simulator $\mathcal{S}$ such that for every vector of inputs $\vec{x}$ it holds: $\{\mathcal{S}(I, \vec{x}_I, f_I(\vec{x}))\} \equiv \{\mathsf{view}^{\pi}_{\mathcal{A}}(\vec{x})\}$, where $\vec{x}_I$ is the inputs of the parties in I, $f_I(\vec{x})$ is the output intended to the parties in I, and $\mathsf{view}^{\pi}_{\mathcal{A}}(\vec{x})$ is $\mathcal{A}$'s view in a real execution of π.*

Remark 4.2 (Straggler resilience) *1.* Round vs. epoch. *Our protocol constructions have a very specific structure, common to concretely efficient n-party computation protocols (à la DN [25]), where execution is divided into phases, or "epochs." In each epoch, a fixed designated party sends messages to the other parties, and then receives back messages from the parties. Within such structure, a somewhat more natural notion of straggler resilience will correspond to a given number of dropped messages per* epoch *(i.e., 2 rounds). However, our notion of τ dropped messages per round is more generally applicable, while still capturing the setting of bounded number of messages dropped per epoch (in this case 2τ, for the two rounds).*

2. Message vs. node drop. *We choose to model latency behavior as embodied by failure of delivery of individual* messages. *This captures settings where delays are caused by network channels, each exhibiting some distribution of latency. This further shares similarities to the "message omission" model, where messages sent to/from affected parties may never be delivered, as considered in, e.g., [45, 53, 52].*

An alternative approach is to consider temporary node *failures per epoch (as considered in, e.g., [54, 49, 60]). This models settings where delays are caused centrally by the node itself. On one hand, our model can be more fine-grained; on the other hand, failure of a node corresponds to failure of potentially many incoming/outgoing communication messages.*

We remark that achieving straggler resilience against node failures poses a challenge within protocols following a star-topology communication structure as in DN and successors since failure of the designated "central" party prevents forward progression of the protocol. Seeing as this protocol structure lies at the core of concretely efficient n-party protocols to date, it remains an interesting open direction to explore whether such node-straggler resilience notion can additionally be achieved with good concrete efficiency.

Observe that the DN protocol $\Pi_{\mathrm{SH}}^{\mathsf{base}}$ from the previous section is not resilient to any straggler. Since it chooses a set S of $2d+1$ parties *in advance* to carry-out the computation, and then the server cannot proceed without all $2d$ messages arriving to him in each multiplication, then an adversary who chooses to drop the messages of even one party in the set S will cause the execution to get stuck. Note that choosing a different set S in each step will not solve the problem, since the adversary is allowed to adaptively choose a different set in each epoch (not to mention the communication cost incurred by resharing intermediate values to the new set of parties).

Next, consider a protocol, where we let all the parties participate in the execution and send their $2d$-degree shares of $xy-r$ to P_1, who then uses the *first* $2d$ shares it receives (together with its own share) to compute $xy-r$. Then, P_1 shares $xy-r$ to the parties, with the optimization outlined above, which allows him to send shares to $n-d-1$ parties only (d shares are always 0).

Note that now the cost is $\frac{n-1+n-d-1}{n} = 2-\frac{d+2}{n}$ field elements sent per party. We denote by $\Pi_{\mathrm{SH}}^{\mathsf{single}}$ a protocol that is identical to $\Pi_{\mathrm{SH}}^{\mathsf{base}}$, with the difference that the input is shared to *all* the parties and multiplication operations are carried-out in the way described above. While the communication cost of $\Pi_{\mathrm{SH}}^{\mathsf{single}}$ is higher than of $\Pi_{\mathrm{SH}}^{\mathsf{base}}$, it does allow $(n-2d-1)$ messages in each epoch to be dropped, since P_1 needs only $2d$ shares in order to compute its message to the parties. For the input sharing and output reconstruction steps, note that $d+1$ shares suffices to compute shared secrets, and so even if $(n-2d-1)$ messages are dropped, there are enough shares to proceed. We thus have:

Theorem 4.3 *Let f be a n-ary functionality over a finite field $\mathbb{F}$ represented by a layered SLP, let t be a security threshold, let d be a parameter such that $d \geq t$, $n \geq 2d+1$ and $|\mathbb{F}| > n+d+1$. Then, Protocol $\Pi_{\mathrm{SH}}^{\mathsf{single}}$ computes f in the $\mathcal{F}_{\mathrm{LinRand}}$-hybrid model, with t-semi-honest-security, $(n-1-2d)$-stragglers-resilience and communication of $2-\frac{d+2}{n}$ field elements sent per party for each bilinear instruction.*

Observe that setting the d parameter gives rise to trade-offs between communication cost, stragglers-resilience and storage cost. Specifically, increasing d reduces communication and also the amount of PRSS keys needed for producing the correlated randomness (see Section 3). In contrast, keeping d small (e.g., setting $d=t$) provides more room for stragglers.

4.3 Reducing Communication and Computation

In this section, we show how to reduce communication and computation cost while still providing resilience to stragglers. This is achieved by taking the approach of packed secret sharing: encoding ℓ secrets over the same polynomial and evaluating ℓ bilinear instructions together, at the cost of a single instruction. We begin with a construction that is designed for SIMD programs, and then show how to extend our techniques to general programs.

4.3.1 Computing SIMD Programs

A program which evaluates the same sub-program many times in parallel is called a SIMD ("same-instruction-multiple-data") straight-line program. Note that a program P which consists of ℓ copies of the same sub-program can be viewed as a program which evaluates each time *a bundle of ℓ identical instructions*. Following works in this area, our idea is to store the ℓ inputs to each bundle on the same polynomial, reducing both communication and computation by a factor of ℓ.

In more details, let $\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}$ be a protocol which is defined as follows. In the pre-processing, the parties prepare a pair of random sharings $[\![r_1 \cdots r_\ell]\!]_d$ and $[\![r_1 \cdots r_\ell]\!]_{2d}$ for each bundle of ℓ bilinear instructions, where $d \geq t + \ell - 1$. Then, in the online protocol, for each bundle of ℓ input instructions, the party who own the inputs shares it as $[\![x_1 \cdots, x_\ell]\!]_d$. For computing a bundle of ℓ bilinear instructions, let $[\![r_1 \cdots r_\ell]\!]_d$,$[\![r_1 \cdots r_\ell]\!]_{2d}$ be the next unused pair. The parties locally compute $[\![x_1 \cdots x_\ell]\!]_d = \sum_{\omega=1}^{w} a_w \, [\![R_{w,1} \cdots R_{w,\ell}]\!]$ and $[\![y_1 \cdots y_\ell]\!]_d = \sum_{\omega=1}^{w} b_w \, [\![R_{w,1} \cdots R_{w,\ell}]\!]$, compute

$$[\![(x_1 y_1 - r_1) \cdots (x_\ell y_\ell - r_\ell)]\!]_{2d} = [\![x_1 \cdots x_\ell]\!]_d \cdot [\![y_1 \cdots y_\ell]\!]_d - [\![r_1 \cdots r_\ell]\!]_{2d}$$

and send the result to P_1. Party P_1 uses the first $2d$ shares it receives together with its own share to compute $x_1 y_1 - r_1, \ldots, x_\ell y_\ell - r_\ell$ and reshares it to the parties as $[\![(x_1 y_1 - r_1) \cdots (x_\ell y_\ell - r_\ell)]\!]_d$. Finally, the parties locally compute

$$[\![x_1 y_1 \cdots x_\ell y_\ell]\!]_d = [\![(x_1 y_1 - r_1) \cdots (x_\ell y_\ell - r_\ell)]\!]_d + [\![r_1 \cdots r_\ell]\!]_d \,.$$

To receive a bundle of ℓ outputs, the parties send their shares to the party who should receive the outputs. This party can then reconstruct the ℓ secrets and obtain its output.

Theorem 4.4 *Let f be a n-ary functionality over a finite field $\mathbb{F}$ represented by a layered SIMD straight-line program P, with bundle of instructions of size ℓ, let t be a security threshold and let d be a parameter, such that $d \geq t + \ell - 1$, $n \geq 2d + 1$ and $|\mathbb{F}| > n + d + \ell + 1$. Then, $\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}$ compute f in the $\mathcal{F}_{\mathrm{LinRand}}$-hybrid model, with t-semi-honest-security, $(n - (2d + 1))$-stragglers-resilience and communication of $\frac{2}{\ell} - \frac{d+2}{n \cdot \ell}$ field elements sent per party for each bilinear instruction.*

Security follows from the same argument as for the base protocol. For stragglers resilience, observe that P_1 can proceed as soon as it holds $2d$ messages, which means that $n - 1 - 2d$ can be dropped. Finally, for each bundle of ℓ instructions, the parties send $n - 1 + (n - 1 - d)$ elements, and so per a single instruction, each party sends $\frac{2n-d-2}{n \cdot \ell} = \frac{2}{\ell} - \frac{d+2}{n \cdot \ell}$ field elements.

Denoting the communication cost per party per instruction of $\Pi_{\mathrm{SH}}^{\mathsf{base}}$ and $\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}$ by $|\Pi_{\mathrm{SH}}^{\mathsf{base}}|$ and $|\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}|$ respectively, we have

$$|\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}| = \frac{|\Pi_{\mathrm{SH}}^{\mathsf{single}}|}{\ell} + \frac{\ell - 1}{n \cdot \ell} < \frac{|\Pi_{\mathrm{SH}}^{\mathsf{single}}|}{\ell} + \frac{1}{n}$$

which since $n \geq 5$ when $\ell \geq 2$, implies that, for computing SIMD bilinear programs, $\Pi_{\mathrm{SH}}^{\mathsf{SIMD}}$ improves communication roughly by a factor of ℓ compared to $\Pi_{\mathrm{SH}}^{\mathsf{single}}$.

4.3.2 Computing General Layered Straight-Line Programs

We next show how use packing to reduce cost when computing any straight-line program. In the protocol, the parties will process in each round ℓ instructions together at the cost of evaluating a

single instruction. For a general-structured program this clearly raises several difficulties. Recall that an instruction in our program consists of taking a linear combination of two sets of inputs and multiply them together. The goal is to carry-out this by packing the "left" inputs on one polynomial and the "right" inputs on a second polynomial and multiply them together, to obtain a polynomial encoding the outputs of ℓ instructions. However, it is now not clear how to proceed to the next batch of ℓ instructions. In particular, when we move from one batch of instructions to the next, the outputs should be reorganized into new blocks of inputs corresponding to the ordering of the inputs in the next ℓ instructions. Moreover, it is possible that an output is used as an input to more than one instruction in the next batch. In this case, we need to ensure that the same value appears in several blocks and possibly in different positions. We call this ordering the "repetition pattern" induced by the program (and define it more formally below). To overcome this challenge, we leverage the fact that in the semi-honest multiplication protocol, party P_1 sees all outputs in the clear, masked using random values. Thus, we can ask P_1 to reshare all values according the ordering of the next batch of instructions. Moreover, to achieve free-addition, we will ask P_1 to first compute the linear combinations over the masked outputs and only then reshare it to the other parties in blocks. The parties, who receive block of masked values, will unmask these values, using correlated randomness they hold, and proceed to the multiplication operation.

Generating correlated randomness according to a repetition pattern. In the protocol, we will need four types of random correlations: (I) $[\![r_1 \cdots r_\ell]\!]_{2d}$ for the output block of each batch of instructions; (II) $[\![r_1, \cdots, r_\ell]\!]_d$ for the left and right input block of each batch of instructions; (III) $[\![r \cdots r]\!]_d$ for masking each party's input; and (IV) $[\![r \cdots r]\!]_d$ for unmasking each party's output.

The "repetition pattern" induces constraints on these random sharings. In particular, for type I, the secrets in each position are unique and independent. The same applies for type III, as the encoded secret is unique and independent from any secret defined for type I. However, for types II and IV, the random secrets are correlated with the other two types. Specifically, at entry k of the encoded block of type II, it must hold that r_k is a linear combination of random secrets that were already defined, according to the structure of the program. For type IV, the encoded random secret should equal to the secret that was chosen for an entry in a block of type I that holds the output. Fortunately, our pre-processing protocol from Section 3 can produce these types of correlated random sharings.

Evaluating a bilinear instruction. In our protocol, the parties hold a sharing of two blocks of ℓ inputs: $[\![x_1 \cdots x_\ell]\!]_d$ and $[\![y_1 \cdots y_\ell]\!]_d$. As in the DN protocol, they locally multiply their shares and add shares of a random block $[\![r_1 \cdots r_\ell]\!]_{2d}$ to obtain a sharing $[\![(x_1 \cdot y_1 + r_1) \cdots (x_\ell \cdot y_\ell + r_\ell)]\!]_{2d}$. Then, the parties send their shares to P_1 who reconstructs $x_1 \cdot y_1 + r_1, \ldots, x_\ell \cdot y_\ell + r_\ell$. However, instead of sending these back to the parties, we let P_1 proceed to the next batch of instructions and compute the linear combinations of the inputs over the masked secrets. Only then P_1 shares the left block of masked inputs and right block of masked inputs to the parties, to perform the next multiplication operation. Once the shares of the blocks of masked inputs are received from P_1, the parties unmask these by adding a block of shared random secret that correspond to the repetition pattern. That is, if we have in the kth position of, say, the left input, a linear combination $(\sum_{\omega=1}^{w} a_{k,\omega} \cdot R_\omega)$ and the value in R_ω was masked using r_ω, then the parties need here a sharing $[\![r`_1 \cdots r`_\ell]\!]_d$ where $r`_k = (\sum_{\omega=1}^{w} a_{k,\omega} \cdot r_\omega)$. As explained previously, our pre-processing protocol from Section 3 can produce these types of random blocks. As before, P_1 proceed once $2d$ shares

have been received, which means that, as before, the protocol is resilient to $n-1-2d$ stragglers. We stress that our trick to let P_1 compute the linear operations over the masked inputs and only then reshare it back to parties, is crucial for achieving addition for free - a property that is not trivial to achieve for non-SIMD circuits.

Sharing the inputs. To share the inputs at the first step of the protocol, if party P_i's input x_i should be masked by r according to the repetition pattern, then the parties generate in the pre-processing a sharing $[\![r \cdots r]\!]_d$ and open it towards P_i. Then, party P_i sends $\hat{x}_i = x_i + r$ to all parties, who use it to produce the shares of masked inputs in future instructions (note that we require sending $\hat{x}_i$ to all parties, and not only to P_1, to achieve stragglers resilience in this step as well).

Formal description and cost analysis. We formally describe our semi-honest protocol in Protocol 4.6. Note that for each batch of ℓ bilinear instruction, $n-1$ parties send an element to P_1, whereas P_1 need to share the inputs of the two inputs blocks, thus sending $2(n-1-d)$ elements. Overall, per a single instruction, each party sends $\frac{n-1+2(n-1-d)}{n\cdot\ell} = \frac{3}{\ell} - \frac{2d+3}{n\cdot\ell}$ field elements, where $d \geq t+\ell-1$.

Theorem 4.5 *Let f be a n-party functionality over a finite field $\mathbb{F}$ represented by a ℓ-layered SLP, let t be a security threshold parameter and let d be a parameter such that $d \geq t+\ell-1$, $n \geq 2d+1$ and $|\mathbb{F}| > n+d+\ell+1$. Then, Protocol 4.6 computes f in the $\mathcal{F}_{\text{LinRand}}$-hybrid model with t-semi-honest-security, $(n-(2d+1))$-stragglers-resilience and communication of $\frac{3}{\ell} - \frac{2d+3}{n\cdot\ell}$ field elements sent per party for each bilinear instruction.*

Proof: Stragglers resilience and communication cost are explained in the text above and thus we move to prove that the protocol is secure in the presence of t semi-honest corrupted parties. Let $\mathcal{S}$ be the ideal world adversary and let $\mathcal{A}$ be the real world semi-honest adversary controlling t parties. $\mathcal{S}$ receives the inputs and outputs of the corrupted parties controlled by $\mathcal{A}$ and needs to simulate their view in the interaction with the honest parties. $\mathcal{S}$ begins by choosing a random tape for the corrupted parties from which their shares of all pre-processed data are derived (recall that the corrupted parties choose their shares of the random sharings generated by $\mathcal{F}_{\text{LinRand}}$). The corrupted parties' random tapes are added to $\mathcal{A}$'s view. We then consider two cases.

Case 1: P_1 is corrupted. In this case, the view of the adversary consists of masked inputs, masked shares of outputs of each multiplication operation and shares for reconstructing the outputs. Thus, $\mathcal{S}$ chooses random elements in the field for the masked inputs and add them to $\mathcal{A}$'s view. For the bilinear instructions, $\mathcal{S}$ chooses random $2d$-degree polynomials, given the corrupted parties' shares (known to $\mathcal{S}$, since it knows the corrupted parties inputs and randomness) and adds the honest parties' shares to $\mathcal{A}$'s view. Observe that until (and not including) the output reconstruction step, the views of $\mathcal{A}$ in the real and simulated execution are identically distributed. This is due to the randomness that we use for masking. Finally, given that all messages are identically distributed in both executions up to the final step, it follows that the shares $\mathcal{A}$ holds of the outputs are distributed the same in both executions, which means that $\mathcal{S}$ can choose the honest parties' shares so that the shares each corrupted party receives will open to the correct output.

PROTOCOL 4.6 (Computing a layered SL Program with Semi-honest Security)

The parties $P_1, \ldots, P_n$ hold a description of a layered straight-line program over $\mathbb{F}$, with m bilinear instructions partitioned to batches of ℓ instructions, such that the inputs to each batch depends only on previous batches. Let ϕ_P be the repetition pattern induced by P.

- **Pre-processing**: The parties call $\mathcal{F}_{\text{LinRand}}$ to obtain sharings $[\![r_1 \cdots r_\ell]\!]_{2d}$ and $[\![r_1 \cdots r_\ell]\!]_d$ for the output block and inputs blocks of each batch of bilinear instructions and $[\![r \cdots r]\!]_d$ for each input/output of P respectively. Theses sharings satisfy the correlation constraints induced by ϕ_P.
- **The Protocol**: The parties emulate the program's instructions as follows:
  1. *Load an input to memory*: For each instruction $R_j \leftarrow x_i$, with x_i held by P_i and $[\![r \cdots r]\!]_d$ being the random sharing that was assigned to the ith input:
     (a) The parties send P_i their shares of $[\![r \cdots r]\!]_d$.
     (b) Party P_i reconstructs r and sends $\hat{x}_i = x_i + r$ to all parties.
  2. *Evaluating the jth batch of bilinear instructions*:
     Let $(\sum_{\omega=1}^{w} a_{1,\omega} \cdot R_\omega) \cdots (\sum_{\omega=1}^{w} a_{\ell,\omega} \cdot R_\omega)$ be the block of left inputs and let $(\sum_{\omega=1}^{w} b_{1,\omega} \cdot R_\omega) \cdots (\sum_{\omega=1}^{w} b_{\ell,\omega} \cdot R_\omega)$ be the block of right inputs.
     For each R_ω for which $\exists a_{1,\omega}, \ldots, a_{\ell,\omega}, b_{1,\omega}, \ldots, b_{\ell,\omega} \neq 0$, party P_1 holds $R_\omega + r_\omega$. Then:
     (a) For $k = 1$ to ℓ: party P_1 locally computes $\sum_{\omega=1}^{w} a_{k,\omega} \cdot (R_\omega + r_\omega)$ and $\sum_{\omega=1}^{w} b_{k,\omega} \cdot (R_\omega + r_\omega)$.
     (b) P_1 shares the block $(\sum_{\omega=1}^{w} a_{1,\omega} \cdot (R_\omega + r_\omega)) \cdots (\sum_{\omega=1}^{w} a_{\ell,\omega} \cdot (R_\omega + r_\omega))$ and the block $(\sum_{\omega=1}^{w} b_{1,\omega} \cdot (R_\omega + r_\omega)) \cdots (\sum_{\omega=1}^{w} b_{\ell,\omega} \cdot (R_\omega + r_\omega))$ to the other parties via a polynomial of degree d.
     (c) The parties locally compute

$$[\![x_1 \cdots x_\ell]\!]_d = \left[\!\!\left[\left(\sum_{\omega=1}^{w} a_{1,\omega} \cdot R_\omega\right) \cdots \left(\sum_{\omega=1}^{w} a_{\ell,\omega} \cdot R_\omega\right) \right]\!\!\right]_d$$
$$= \left[\!\!\left[\left(\sum_{\omega=1}^{w} a_{1,\omega} \cdot (R_\omega + r_\omega)\right) \cdots \left(\sum_{\omega=1}^{w} a_{\ell,\omega} \cdot (R_\omega + r_\omega)\right) \right]\!\!\right]_d - [\![r'_1 \cdots r'_\ell]\!]_d$$

     and

$$[\![y_1 \cdots y_\ell]\!]_d = \left[\!\!\left[\left(\sum_{\omega=1}^{w} b_{1,\omega} \cdot R_\omega\right) \cdots \left(\sum_{\omega=1}^{w} b_{\ell,\omega} \cdot R_\omega\right) \right]\!\!\right]_d$$
$$= \left[\!\!\left[\left(\sum_{\omega=1}^{w} b_{1,\omega} \cdot (R_\omega + r_\omega)\right) \cdots \left(\sum_{\omega=1}^{w} b_{\ell,\omega} \cdot (R_\omega + r_\omega)\right) \right]\!\!\right]_d - [\![r''_1 \cdots r''_\ell]\!]_d$$

     where $[\![r'_1 \cdots r'_\ell]\!]_d = [\![(\sum_{\omega=1}^{w} a_{1,\omega} \cdot r_\omega) \cdots (\sum_{\omega=1}^{w} a_{\ell,\omega} \cdot r_\omega)]\!]_d$ and $[\![r''_1 \cdots r''_\ell]\!]_d = [\![(\sum_{\omega=1}^{w} b_{1,\omega} \cdot r_\omega) \cdots (\sum_{\omega=1}^{w} b_{\ell,\omega} \cdot r_\omega)]\!]_d$ were produced for these blocks in the per-processing.
     (d) The parties locally compute

$$[\![(x_1 \cdot y_1 - r_1) \cdots (x_\ell \cdot y_\ell - r_\ell)]\!]_{2d} = [\![x_1 \cdots x_\ell]\!]_d \cdot [\![y_1 \cdots y_\ell]\!]_d - [\![r_1 \cdots r_\ell]\!]_{2d}$$

     where $[\![r_1 \cdots r_\ell]\!]_{2d}$ is the block of random shares produced for this batch in the pre-processing step.
     (e) The parties send their shares to P_1 who reconstruct $z_1, \ldots, z_\ell$ where $\forall k \in [\ell]: \; z_k = x_k \cdot y_k + r_k$ and store the result.
     (f) If z_k is an output of the program for some $k \in [\ell]$, then P_1 shares z_k to the parties using a d-degree polynomial. Then, the parties unmask it by locally computing $[\![z_k \cdots z_k]\!]_d - [\![r \cdots r]\!]_d$, where r is the mask used in the output block where z_k was computed.
  3. *Output value from memory:* for each instruction $O_i \leftarrow R_j$, where P_i should receive O_i, the parties send their shares of the value in R_j to P_i who reconstruct and output O_i.

Case 2: P_1 is honest. In this case, the view of the adversary during the execution consists only of messages sent to it by P_1 for the block inputs for each multiplication operation. To simulate these, $\mathcal{S}$ chooses random d-degree polynomials for each input block and add the corrupted parties' shares to $\mathcal{A}$'s view. Since each secret is masked by a random element, it follows that the view is identically distributed in both executions. For the output reconstruction step, the proof is the same as in the previous case. This concludes the proof. ■

Observe that when $\ell \geq 3$ (i.e., packing at least 3 secrets on each polynomial), we have $\frac{3}{\ell} - \frac{2d+3}{n \cdot \ell} < 1$, which means that each party sends *less than one field element* for each bilinear instruction. When $\ell = 2$, then the cost is less than 1.5 elements sent per party. We thus obtain a protocol which provide the best of both worlds: it achieves both minimal communication and stragglers resilience. This is in contrast to $\Pi_{\text{SH}}^{\mathsf{base}}$ which achieves minimal communication without any resilience to stragglers, and $\Pi_{\text{SH}}^{\mathsf{single}}$ which can handle stragglers but at the cost of (at least) doubling the communication cost. We provide exact cost analysis with concrete numbers below. Finally, we stress that the only assumption that our protocol makes on the structure of program, is that it is possible to split the program into batches of ℓ instructions where the input to each instruction in this batch comes from batches that precede it (this property is reflected in the "ℓ-layered SLP" notion). This is a mild assumption that is satisfied by natural circuits.

4.4 Concrete Efficiency Analysis

In this section, we analyze the efficiency of our protocol, by looking at the communication cost (measured by the number of field elements sent per multiplication instruction) and storage/computation cost (measured by the number of PRSS seeds and number of PRF invocations per multiplication instruction).

Communication cost. We first discuss the exact communication cost of our protocols for concrete parameters. The exact cost depends on several parameters: the security threshold t (i.e., number of corrupted parties), number of stragglers τ allowed in each epoch and the packing parameter ℓ. For this analysis, we assume $d = t + \ell - 1$.

In Table 2 we present the total communication cost (as number of field elements) sent per multiplication instruction/gate for different combinations of parameters. For each combination, we present the resulted number of parties n, which equals to $2d+1+\tau = 2(t+\ell-1)+1+\tau = 2t+2\ell-1+\tau$, the number of total field elements sent (by all parties) and the ratio between this and the minimal number of elements sent when no resilience to stragglers and no secret packing is considered. When $\ell = 1$, the parties send $n-1$ elements to P_1 to receive back $n-1-t$ elements. Thus, the total number of elements is $2(n-1) - t = 3t + \tau$. When $\ell \geq 2$, we use the formula from Theorem 4.5, namely, $\frac{3n-2t-2\ell-1}{\ell}$. For the communication cost without stragglers resilience, we use $\Pi_{\text{SH}}^{\mathsf{base}}$ from Section 4.1 (where the parties share their inputs to a committee of $2t+1$ parties) and so the total number of field elements is exactly $3t$.

The main observation from the table is that, as we increase the corruption threshold t and the packing parameter ℓ, we are able to beat the baseline protocol while also providing resilience to stragglers. In particular, when $t = 1$ the baseline protocol requires less communication than our protocol in each of the examined set of parameters. When $t = 2$, our protocol starts to beat the baseline protocol when $\ell = 4$. When $t = 4$, our protocol beats the baseline protocol when $\ell = 2$ and $\tau = 1$, and in all cases for $\ell = 4$ and $\ell = 8$. Observe also that the improvement upon the base

# Corruptions (t)	# Stragglers (τ)	$\ell = 1$	$\ell = 2$	$\ell = 4$	$\ell = 8$
1	0	$n = 3$: 3	$n = 5$: 4.0 (×1.33)	$n = 9$: 4.0 (×1.33)	$n = 17$: 4.0 (×1.33)
	1	$n = 4$: 4 (×1.33)	$n = 6$: 5.5 (×1.83)	$n = 10$: 4.8 (×1.6)	$n = 18$: 4.4 (×1.46)
	2	$n = 5$: 5 (×1.67)	$n = 7$: 7 (×2.33)	$n = 11$: 5.5 (×1.83)	$n = 19$: 4.8 (×1.58)
	3	$n = 6$: 6 (×2)	$n = 8$: 8.5 (×2.83)	$n = 12$: 6.3 (×2.1)	$n = 20$: 5.1 (×1.71)
2	0	$n = 5$: 6	$n = 7$: 6.0 (×1)	$n = 11$: 5.0 **(×0.83)**	$n = 19$: 4.5 **(×0.75)**
	1	$n = 6$: 7 (×1.16)	$n = 8$: 7.5 (×1.25)	$n = 12$: 5.8 **(×0.96)**	$n = 20$: 4.9 **(×0.81)**
	2	$n = 7$: 8 (×1.32)	$n = 9$: 9 (×1.5)	$n = 13$: 6.5 (×1.08)	$n = 21$: 5.3 **(×0.88)**
	3	$n = 8$: 9 (×1.5)	$n = 10$: 10.5 (×1.75)	$n = 14$: 7.3 (×1.21)	$n = 22$: 5.6 **(×0.94)**
3	0	$n = 7$: 9	$n = 9$: 8.0 **(×0.89)**	$n = 13$: 6.0 **(×0.67)**	$n = 21$: 5.0 **(×0.56)**
	1	$n = 8$: 10 (×1.11)	$n = 10$: 9.5 (×1.05)	$n = 14$: 6.8 **(×0.75)**	$n = 22$: 5.4 **(×0.6)**
	2	$n = 9$: 11 (×1.22)	$n = 11$: 11 (×1.22)	$n = 15$: 7.5 **(×0.83)**	$n = 23$: 5.8 **(×0.64)**
	3	$n = 10$: 12 (×1.33)	$n = 12$: 12.5 (×1.38)	$n = 16$: 8.3 **(×0.92)**	$n = 24$: 6.1 **(×0.68)**
4	0	$n = 9$: 12	$n = 11$: 10 **(×0.83)**	$n = 15$: 7.0 **(×0.58)**	$n = 23$: 5.5 **(×0.45)**
	1	$n = 10$: 13 (×1.08)	$n = 12$: 11.5 **(×0.96)**	$n = 16$: 7.8 **(×0.65)**	$n = 24$: 5.9 **(×0.49)**
	2	$n = 11$: 14 (×1.17)	$n = 13$: 13 (×1.08)	$n = 17$: 8.5 **(×0.71)**	$n = 25$: 6.3 **(×0.52)**
	3	$n = 12$: 15 (×1.25)	$n = 14$: 14.5 (×1.21)	$n = 18$: 9.3 **(×0.77)**	$n = 26$: 6.6 **(×0.55)**
8	0	$n = 17$: 24	$n = 19$: 18 **(×0.75)**	$n = 23$: 11.0 **(×0.46)**	$n = 31$: 7.5 **(×0.31)**
	1	$n = 18$: 25 (×1.04)	$n = 20$: 19.5 **(×0.81)**	$n = 24$: 11.75 **(×0.48)**	$n = 32$: 7.88 **(×0.33)**
	2	$n = 19$: 26 (×1.08)	$n = 21$: 21 **(×0.88)**	$n = 25$: 12.5 **(×0.52)**	$n = 33$: 8.25 **(×0.34)**
	3	$n = 20$: 27 (×1.13)	$n = 22$: 22.5 **(×0.94)**	$n = 26$: 13.25 **(×0.55)**	$n = 34$: 8.63 **(×0.36)**

Table 2: Total number of field elements sent per multiplication as a function of the number of corrupted parties (t), packing parameter (ℓ) and number of stragglers (τ). For each combination we show: (i) the number of parties, computed as $n = 2t + 2\ell - 1 + \tau$; (ii) total number of elements sent per multiplication, computed via the formula $3t + \tau$ for $\ell = 1$ and $\frac{3n-2t-2\ell-1}{\ell}$ for $\ell \geq 2$; and (iii) the ratio between (ii) and the best cost of a similar protocol with $n = 2t + 1$ parties and no straggler resilience or secret packing. The latter is equal to $3t$. Boldface entries are ones in which our protocol has better concrete *total* communication complexity, while typically also tolerating stragglers (when $\tau \geq 1$).

protocol also grows as we increase t and ℓ, reaching a factor of approximately 2 when $t = 4$ and $\ell = 8$. Finally, when $t = 8$, our protocol beats the baseline protocol even when $\ell = 2$.

Storage and computation cost for SIMD circuits. We proceed to show the number of PRSS seeds each party needs to store and the number of calls to a PRF per party to produce the double sharing required for multiplying shared inputs in an evaluation of a SIMD circuit. In Table 3 we present a comparison between our construction and CDI [21], for various number of corruptions (t) and stragglers (τ), and for $\ell = 8$ (where ℓ is the packing parameter, which in SIMD circuits is also the number of sub-circuits that are being evaluated in parallel). Note that in this setting, we have $d = t + \ell - 1 = t + 7$ and $n = 2d + 1 = 2t + 15$. Note also that by Theorem 4.4, the total number of elements sent by each party per multiplication instruction is less than $\frac{2}{\ell}$, and so in our example less than 0.25 field elements. For our construction, we use Theorem 3.6 in Section 3.5 to compute the number of seeds and PRF calls per party. For the CDI construction, the number of seeds is $\binom{n-1}{t}$. To compute the number of PRF invocations per multiplication instruction, we used the same method as in Section 3.5, namely, to generate 2 polynomials - one with degree-d and one with degree $2d - \ell$- and the trick in footnote 2. This implies that the number of calls to a PRF is $\binom{n-1}{t} \cdot (\ell + (2d - \ell - t)) = \binom{n-1}{t} \cdot (2d - t)$. As can be seen from the table, as t grows, we reduce dramatically both storage costs and the computational overhead.

# Corruptions (t)	# Stragglers (τ)	# Parties (n)	Best known cover size $(n, d+1, t)$	Best known cover size $(n, 2d-\ell+1, t)$	Ours # seeds and PRF calls per party	CDI [21] # seeds per party	CDI [21] # PRF calls per party
1	0	17	2	2	20	16	240
	1	18	2	2	20	17	255
	2	19	2	2	21	18	270
	3	20	2	2	22	19	285
2	0	19	6	5	58	153	2,448
	1	20	6	6	66	171	2,736
	2	21	7	6	75	190	3,040
	3	22	7	6	78	210	3,360
4	0	23	31	15	283	7,315	131,670
	1	24	31	18	314	8,855	159,390
	2	25	47	21	455	10,626	191,268
	3	26	51	24	520	12,650	227,700
8	0	31	2,628	45	22,003	5.8×10^6	1.3×10^8
	1	32	3,302	81	28,650	7.8×10^6	1.73×10^8
	2	33	5,211	121	46,406	1.05×10^7	2.3×10^8
	3	34	6,613	176	60,557	1.4×10^7	3.05×10^8

Table 3: Storage cost (number of PRF seeds per party) and computation cost (number of calls to PRF per party per multiplication instruction) for SIMD circuits with $\ell = 8$ (recall that ℓ is the packing parameter, which in SIMD circuits can be viewed as the number of sub-circuits that are being evaluated in parallel), as a function of number of corrupted parties (t) and number of stragglers (τ). The number of parties is computed by taking $n = 2d + 1$, where $d = t + \ell - 1$. The cover size, k' for $(n, d+1, t)$-covers and k'' for $(n, 2d-\ell+1, t)$-covers, is taken from [1]. By Theorem 3.6, the number of seeds per party is $(k'(d+1)(n-d) + k''(d+t)(n-d-t+1))/n$ and there is one PRF invocation per seed. The construction in Appendix B is an alternative with a different tradeoff between storage and PRF invocations. For the CDI construction, the number of seeds per party is $\binom{n-1}{t}$ and number of PRF calls is $\binom{n-1}{t} \cdot (2d-t)$. See Table 2 for concrete communication costs.

Storage and computation for general circuits. So far, we only considered the case where $d = t + \ell - 1$. When working over general (non-SIMD) circuits, this implies that each party needs to store $\binom{n-1}{t}$ pseudorandom seeds in order to produce the correlated randomness for our protocol (since our PRSS construction in Section 3.6 uses a $(n, d-\ell+1, t)$-cover). However, we can reduce this cost by increasing d. Specifically, given t, ℓ and τ, it is possible to choose d such that $d > t + \ell - 1$, and then we have $n = 2d + 1 + \tau$. In this case, the communication cost for each multiplication instruction per party is $\frac{3}{\ell} - \frac{2d+3}{n \cdot \ell}$ according to Theorem 4.5. The amount of PRF seeds stored by each party, according to Theorem 3.7, is $k'(d-\ell+1)(n-d+\ell)/n$, where k' is a size of an $(n, d-\ell+1, t)$-cover.

For example, consider $t = 4$, $\ell = 3$, $d = 18$ and $\tau = 11$. In this case, we need $n = 48$ parties. The best cover size known for this setting (see Table 1) is 252, which implies that the amount of seeds per party is 2268. The communication cost per party in this setting is just $1 - \frac{13}{48} \approx 0.73$ field elements per multiplication. On the other hand, when using $d = t + \ell - 1 = 6$, we have $n = 24$. In this case, the communication cost per party is $1 - \frac{5}{24} = 0.79$ elements and the storage cost per party is 10,626 seeds.

Our analysis clearly shows the potential of all our methods together to significantly improve the efficiency of secure computation with strong honest majority.

5 From Semi-Honest to Malicious Security

In this section, we show how to augment our protocol from the previous section to malicious security (with abort). Our goal is to achieve malicious security without increasing the amortized communication cost per instruction, and while maintaining the resilience to stragglers.

We begin by defining the meaning of security and resilience to stragglers in the presence of malicious adversaries. Note that unlike the definition with semi-honest adversaries, we no longer guarantee a successful termination of the protocol, but rather provide security with abort. The straggler-robust correctness, however, will still require that the protocol ends successfully if the parties act honestly, even if in each round τ messages, chosen by the adversary, are dropped. In addition to this requirement, we also need the protocol to be secure in the presence of an adversary who controls t parties and, in addition, can drop any τ messages in each round of communication.

Following the standard ideal-world vs. real-world paradigm of MPC [33, 15], let $\mathcal{A}$ be an adversary who chooses a set of parties before the beginning of the execution and corrupts them. We assume that the adversary is *rushing*, meaning that it first receives the messages sent by the honest parties in each round, and only then determines the corrupted parties' messages in this round. Let $\text{REAL}^{f}_{\Pi,\mathcal{A},I}(1^\kappa, \vec{x})$ be a random variable that consists of the view of the adversary $\mathcal{A}$ controlling a set of parties I, and the honest parties' outputs, following an execution of Π over a vector of inputs $\vec{x}$ to compute f with security parameter κ. Similarly, we define an ideal-world execution with an ideal-world adversary $\mathcal{S}$, where $\mathcal{S}$ and the honest parties interact with a trusted party who computes f for them. We consider secure computation *with abort*, meaning that $\mathcal{S}$ is allowed to send the trusted party computing f a special command abort. Specifically, $\mathcal{S}$ can send an abort command instead of handing the corrupted parties' inputs to the trusted party (causing all parties to abort the execution), or, hand the inputs and then, after receiving the corrupted parties' outputs from the trusted party, send the abort command, and prevent them from receiving their outputs. We denote by $\text{IDEAL}_{f,\mathcal{S},I}(1^\kappa, \vec{x})$, the random variable that consists of the output of $\mathcal{S}$ and the honest parties in an ideal execution to compute f, over a vector of inputs $\vec{x}$, where $\mathcal{S}$ controls a set of parties I. The security definition states that a protocol Π securely computes f with statistical error ε, if for every real-world adversary there exists an ideal-world adversary, such that the statistical distance between the two random variables is less than ε.

Definition 5.1 (Straggler resilience, malicious security) *Let f be an n-party functionality and let $\varepsilon = \varepsilon(\kappa)$ be a statistical error bound. We say that Π computes f with t-malicious-security-with-abort and τ-straggler-resilience with statistical error ε if it satisfies the following properties:*

- Straggler-Robust Correctness: *If all parties act honestly, then Π terminates successfully (i.e. each party receives its prescribed output $f_i(\vec{x})$) even if in each communication round, τ messages, chosen adaptively by the adversary, are not delivered.*
- Security with Stragglers: *For every real-world malicious adversary $\mathcal{A}$ who controls a set of parties I with $|I| \leq t$ and, in addition, can choose adaptively any τ messages to drop in each round of communication, there exists an ideal-world simulator $\mathcal{S}$, such that for every κ and every vector of inputs $\vec{x}$ it holds that*

$$SD\left(\text{REAL}^{f}_{\Pi,\mathcal{A},I}(1^\kappa, \vec{x}), \text{IDEAL}_{f,\mathcal{S},I}(1^\kappa, \vec{x})\right) \leq \varepsilon$$

where $SD(X,Y)$ is the statistical distance between X and Y[4].

To construct a protocol that satisfies the definition, we work in two steps. First, we present a protocol to compute the program until (and not including) the output-revealing stage, that provides privacy in the presence of malicious adversaries. As we will see, maybe somewhat contrary to intuition, our semi-honest protocol from the previous section may leak private data to a malicious adversary. We thus show how to fix this without changing the communication cost or the round complexity and whilst providing the same resilience to stragglers.

Then, we add a step, before the revealing of the output, in which the parties verify the correctness of the computation, and abort with high probability if cheating took place. The properties of this step are: (i) it has sublinear communication (in the size of the program) and so the overall amortized communication cost per instruction remains the same, (ii) it requires a small constant number of rounds and so does not increase the round complexity of our protocol.

We note that although the protocol we describe only guarantees security with *selective* abort, it can be easily augmented to *unanimous* abort as required by the definition above with small constant cost, by running a single Byzantine agreement before the end of the execution. For simplicity, we omit this step from the description.

Before proceeding, we briefly describe two building blocks required by our protocol:

The $\mathcal{F}_{\text{coin}}$ ideal functionality. In our protocol, the parties will sometimes need to produce fresh random coins. The $\mathcal{F}_{\text{coin}}$ functionality, when called by the parties, hands them such coins. To compute $\mathcal{F}_{\text{coin}}$ with abort, the parties can simply generate a random sharing $[\![r]\!]_d$ and open it. In the honest majority setting, there is nothing the adversary can do here beyond causing an abort. We note that to generate any number of coins with constant communication cost, it suffices to call $\mathcal{F}_{\text{coin}}$ once to obtain a seed, and expand it to many pseudo-random coins.

Consistency check. To check that m sharings $\{[\![x_{j,1} \cdots x_{j,\ell}]\!]_d\}_{j=1}^m$ are consistent, we use the well-known method of taking a random linear combination of these sharings, mask the result by adding a random sharing $[\![r_1 \cdots r_\ell]\!]_d$, and open it. For the random linear combination, the parties call $\mathcal{F}_{\text{coin}}$ to obtain the random coefficients.

5.1 Privacy in the Presence of Malicious Adversaries

In this section, we show how to compute a straight-line program with privacy in the presence of a malicious adversary. We begin by showing that DN-style semi-honest protocols which we consider in this work, may leak private information to a malicious adversary in the strong honest majority setting. Recall that in the semi-honest protocol, to carry-out a multiplication between shared inputs $[\![x]\!]_d$ and $[\![y]\!]_d$, the parties send $[\![x \cdot y - r]\!]_{2d}$ to P_1, who reconstruct $x \cdot y - r$ and shares it as $[\![x \cdot y - r]\!]_d$ to the parties. Then, the parties compute $[\![x \cdot y]\!]_d = [\![x \cdot y - r]\!]_d + [\![r]\!]_d$ and obtain a sharing of the output.

[4] Note that we prove statistical security of our protocol in a hybrid model where parties hold correlated randomness. The resulting combined protocol provides computational security when this setup is instantiated using PRSS.

The "double-dipping" attack [40]. We now describe an attack that can be carried out by a malicious P_1, when $n > 2d+1$. This attack was shown in [40] for the setting of $d < n/3$ and works over two multiplication gates/instructions as follows. Assume that the parties multiply $[\![x]\!]_d$ with $[\![y]\!]_d$. Thus, after receiving the masked shares from the parties, P_1 reconstructs $xy-r$ and computes a random sharing $[\![x \cdot y - r]\!]_d$. Then, P_1 sends the correct shares to all parties except for P_n, to whom it adds 1 to the intended share. Thus, all the parties, except for P_n, can compute the correct share of $x \cdot y$ by adding $[\![r]\!]_d$. Denote the share of $x \cdot y$ held by P_i by α_i. This means that P_n will hold $\alpha_n + 1$. Next, assume that the parties proceed to the next multiplication, where they need to multiply $[\![xy]\!]_d$ with $[\![z]\!]_d$, and denote the share of z held by P_i by z_i. Note that once P_1 receives $2d$ shares, it can not only reconstruct $xyz - r'$, where r' is the random masking for this multiplication, but also can compute the remaining $n-1-2d$ shares that should be sent. In particular, after receiving shares from any subset of $2d$ parties that does not contain P_n, it can compute the correct share that should be sent by P_n, i.e., $\alpha_n \cdot z_n - r'_n$, where r'_n is P_n's share of r'. However, P_n will send the share $(\alpha_n+1) \cdot z_n - r'_n$, which means that P_1 can compute $(\alpha_n \cdot z_n - r'_n) - ((\alpha_n+1) \cdot z_n - r'_n) = z_n$, obtaining the secret share z_n of P_n.

Previous solutions. The main reason for the above attack is that in the strong honest majority setting, there is redundancy in the masking. Indeed, the solution suggested in [40] is to use as masking the sharing $[\![r]\!]_{n-1}$, which means that $x \cdot y - r$ can be reconstructed only given the shares of all parties. A different solution was given in [29], where a consistency check was carried-out between each two layers of the program. This prevents the above attack, since by sending an incorrect share to P_n, the resulting sharing of $x \cdot y$ becomes inconsistent. Thus, a consistency check will detect this type of cheating and prevents P_1 from proceeding with the attack to the multiplication in the next layer. However, these solutions are not sufficient in our case, since either they require all parties to participate, preventing any resilience to stragglers, or, double the round complexity of the protocol.

A new solution with straggler resilience. We thus need a new solution that achieves privacy, while allowing P_1 to proceed without requiring all parties' shares of $x \cdot y - r$. Our idea is to have a *different independent masking value for each subset of* $2d+1$ *parties*. In particular, for each subset T of $2d+1$ parties, we want the parties to hold a pair $([\![r_T]\!]_d, [\![r_T]\!]_{2d})$ which can be used in the multiplication protocol. This however raises a question. If each subset of parties have a different masking, then which masking share should a party use when it sends its message to P_1? To overcome this, we add an additional constraint: the parties should hold a pair $([\![r_T]\!]_d, [\![r_T]\!]_{2d})$ for each subset T under the constraint that each P_i's share in $[\![r_T]\!]_{2d}$ will be *identical* for all subsets. If this holds, then only one possible message exists for each P_i to send to P_1 (i.e., $x_i \cdot y_i - r_i$ where r_i is the random share used by P_i as a mask). We will see later how to generate such correlated randomness in an efficient way (without requiring the parties to store $\binom{n}{2d+1}$ different polynomials). Assuming the parties have a way to generate such random sharings, our private protocol to multiply $[\![x]\!]_d$ and $[\![y]\!]_d$ will work as follows:

$\underline{\Pi^{\mathsf{priv}}_{\mathrm{mult}}}$**:**

- **Inputs**: Each P_i holds two inputs shares x_i, y_i and a random share r_i.
  For each subset $T \subset \{P_1, \ldots, P_n\}$ such that $|T| = 2d+1$, the parties hold a sharing $[\![r_T]\!]_d$, where

$r_T = \sum_{j|P_j \in T} \lambda_j \cdot r_j$, with λ_j being the corresponding Lagrange coefficient for the $2d$-polynomial q_T defined such that $q_T(j) = r_j$, for each j for which $P_j \in T$.

- **The protocol:**

1. Each party P_i locally computes $e_i = x_i \cdot y_i - r_i$ and sends it to P_1.
2. Let $e_{i_1}, \ldots, e_{i_{2d}}$ be the first $2d$ messages received by P_1 and let T be a subset of parties defined as $T = \{P_1, P_{i_1}, \ldots, P_{i_{2d}}\}$. Then, P_1 view $e_1, e_{i_1}, \ldots, e_{i_{2d}}$ as points on a polynomial p of $2d$-degree such that $p(1) = e_1 \quad \mathsf{and} \quad \forall j \in [2d] : p(i_j) = e_{i_j}$ and uses them to compute (via Lagrange interpolation) the value $e_0 = p(0)$.
3. P_1 chooses a new random sharing $[\![e_0]\!]_d$, under the constraint that d shares equal to 0, and sends each party P_i, with a non-zero share, its share. In addition, it sends T to all parties.
4. The parties locally compute $[\![x \cdot y]\!]_d = [\![e_0]\!]_d + [\![r_T]\!]_d$.

It is easy to see that if the parties follow the protocol, then they will obtain $[\![x \cdot y]\!]_d$. Privacy is achieved since now there is no redundancy in the secret sharing of the masking random element, and each random share held by each party is independent from the other parties' random shares. To formally prove this, we need to show that the view of an adversary controlling up to d parties is distributed identically, regardless of the input held by the honest parties. Let $\Pi^{\mathsf{single}}_{\mathrm{priv}}$ be a protocol where the the parties compute a straight-line program as in $\Pi^{\mathsf{single}}_{\mathrm{SH}}$ while carry-out multiplication operations using $\Pi^{\mathsf{priv}}_{\mathrm{mult}}$, namely, each party shares its inputs and then the parties traverse over the program instruction by instruction, while additions are computed locally and multiplications using $\Pi^{\mathsf{priv}}_{\mathrm{mult}}$. In addition, let $\mathsf{view}^f_{\mathcal{A},\Pi,I}(\vec{v})$ be the view of the adversary in the execution of a protocol Π computing a functionality f *without the output revealing step*, when controlling a set I of parties, on a vector of inputs $\vec{v}$.

Theorem 5.2 *Let f be a n-ary functionality represented by a layered straight-line program over a finite field $\mathbb{F}$ and let d be a threshold parameter (such that $|\mathbb{F}| > n + d + 1$). Then, for every adversary $\mathcal{A}$, every subset $I \subset \{P_1, \ldots, P_n\}$ with $|I| \leq d$ and for every two vectors of inputs $\vec{v}_1, \vec{v}_2$, it holds that $\mathsf{view}^f_{\mathcal{A},\Pi^{\mathsf{single}}_{\mathrm{priv}},I}(\vec{v}_1) \equiv \mathsf{view}^f_{\mathcal{A},\Pi^{\mathsf{single}}_{\mathrm{priv}},I}(\vec{v}_2)$.*

Proof: The view of an adversary in $\Pi^{\mathsf{single}}_{\mathrm{priv}}$ consists of (up to) d shares received in the input sharing step, and the view in the execution of $\Pi^{\mathsf{priv}}_{\mathrm{mult}}$. In the former, since each input is shared via a random polynomial of degree d, then any d points look completely random regardless of the shared secret. We thus proceed to show that for any two vector of inputs, the view of the adversary $\mathcal{A}$ in $\Pi^{\mathsf{priv}}_{\mathrm{mult}}$ is identically distributed. We consider two case.

Case 1: P_1 is honest. In this case, in each execution of $\Pi^{\mathsf{priv}}_{\mathrm{mult}}$, $\mathcal{A}$ sees e_0, the subset T used to compute e_0 and for each subset of parties T' of size $2d$, it sees d shares from $[\![r_{T'}]\!]_d$. Recall that

$$\begin{aligned} e_0 &= \lambda_1 \cdot e_1 + \sum_{j=1}^{2d} \lambda_{i_j} \cdot e_{i_j} = \lambda_1 \cdot (x_1 \cdot y_1 - r_1) + \sum_{j=1}^{2d} \lambda_{i_j} \cdot (x_{i_j} \cdot y_{i_j} - r_{i_j}) \\ &= \lambda_1 \cdot (x_1 \cdot y_1) + \sum_{j=1}^{2d} \lambda_{i_j} \cdot (x_{i_j} \cdot y_{i_j}) - r_T. \end{aligned}$$

Let $\alpha = \lambda_1 \cdot (x_1 \cdot y_1) + \sum_{j=1}^{2d} \lambda_{i_j} \cdot (x_{i_j} \cdot y_{i_j})$. Thus, $\mathcal{A}$'s view consists of $\alpha - r_T$ and d points on a d-degree polynomial q, with $q(0) = r_T$. Note that each vector of inputs defines deterministically the value of α. Now, fixing a vector of inputs $\vec{x}$ and $\vec{y}$, we have that the probability for each possible view equals to the probability that $r_T = \alpha_0 - e_0$, which is $\frac{1}{|\mathbb{F}|}$. This hols since r_T is computed using a completely random and independent share of at least one honest party. Since the above holds for any vector of inputs, it follows that the theorem holds for this case.

Case 2: P_1 is corrupted. In this case, $\mathcal{A}$ receives all the messages sent to P_1. Thus, the view of $\mathcal{A}$ consists of $x_i \cdot y_i - r_i$ for all i. In addition, it holds d shares of r_T for each subset T of size $2d+1$. Since d shares gives no information on the values of r_T and since each r_i is completely random and independent from the others, it follows by a similar argument to the previous case, that for each vector of inputs, $\mathcal{A}$'s view is uniformly distributed over $\mathbb{F}$. This completes the proof. ■

Efficient generation of the correlated randomness. Recall that our protocol requires that for each multiplication, each P_i will hold a random independent r_i and a sharing $[\![r_T]\!]_d$ for each subset of parties T of size $2d+1$, such that $r_T = \sum_{j|P_j \in T} \lambda_j \cdot r_j$. A simple way to achieve this, is to let each P_i choose a random r_i and share it to the other parties as $[\![r_i]\!]_d$. Upon holding $[\![r_i]\!]_d$ for each $i \in [n]$, the parties can locally compute $[\![r_T]\!]_d = \sum_{j|P_j \in T} \lambda_j \cdot [\![r_j]\!]_d$ for each subset T of size $2d+1$. We note that in order to save cost, the parties can defer the last step of computing $[\![r_T]\!]_d$ until they receive the subset T from P_1. This is significant since now the parties need to compute just a single sharing of degree d and not $\binom{n}{2d+1}$.

To generate any number of such correlated randomness without any interaction but a short setup step, each party P_i can distribute a set of seeds to the other parties. As explained in Section 3, it is possible to non-interactively generate any number of Shamir's secret sharings $[\![r_i]\!]_d$ from these seeds and then continue as above. Note that since P_i knows all seeds, it can locally compute r_i and use it as its mask in the multiplication operation as required.

Extending the protocol to packed secret sharing. Till now, we explained how to achieve privacy without requiring all parties to participate in each epoch, in the single secret per sharing case. We now explain how to extend the protocol when multiple secrets are packed together. Recall that in this case, the parties compute each time a batch of ℓ bilinear instructions, in which P_1 computes addition operations over masked inputs, share two input blocks to a multiplication operation, and then the parties interact to multiply the two blocks of ℓ inputs. For each such batch of instructions, the requirements are: (i) each party P_i should hold a random independent element r_i, which will be used as a mask in his message to P_1; (ii) for each input block to the multiplication operation, the parties should hold a sharing $[\![r_1 \cdots r_\ell]\!]_d$ which satisfies some linear constraints induced by the repetition pattern of the program. Specifically, for each $k \in [\ell]$, it should hold that $r_k = \sum_{\omega=1}^{w} a_\omega \cdot r_\omega^T$, where r_ω^T is the masking used in some previous multiplication operation. Note that r_ω^T is determined on-the-fly, with T being the subset of $2d$ parties, whose shares were the first to arrive to P_1, when computing the batch of instructions for which r_ω^T was used as a mask.

Relying on the constructions from Section 3, the parties thus work as follows:

- Before the start of the computation: each party P_i distributes seeds to the other parties.

- For the jth batch of ℓ instructions:

  1. For each $i \in [n]$, the parties convert their set of seeds from P_i to $[\![r_{j,i}0\cdots0]\!]_d, [\![0r_{j,i}0\cdots0]\!] \ldots, [\![0\cdots0r_{j,i}]\!]_d$. Observe that P_i knows the value of $r_{j,i}$.
  2. Party P_i uses $r_{j,i}$ as its mask in the message it sends to P_1: let $x_{j,i}$ ($y_{j,i}$) be its share of the left(right) block of ℓ inputs. Then, P_i sends $x_{j,i} \cdot y_{j,i} - r_{j,i}$ to P_1.
  3. Let T_j be a subset of $2d+1$ parties whose messages to P_1 arrived first. Then, for each $k \in [\ell]$: P_1 reconstructs the value $x_{j,k} \cdot y_{j,k} - r_{j,k}^{T_j}$, by computing $\sum_{i|P_i \in T_j} \lambda_{i,k} \cdot (x_{j,i} \cdot y_{j,i} - r_{j,i})$, where $\lambda_{i,k} = \prod_{m \neq i | P_m \in T_j} \frac{(-k+1)-m}{i-m}$ is the Lagrange coefficient corresponding to using points held by parties in T_j to compute the value at the point $(-k+1)$. Note that this implies that $r_{j,k}^{T_j} = \sum_{i|P_i \in T_j} \lambda_{i,k} \cdot r_{j,i}$ is the mask used in this batch of instructions.
  4. Unmasking input blocks: upon receiving a block $[\![x'_1 \cdots x'_\ell]\!]_d$ from P_1, let $x'_k = \sum_{\omega=1}^{w} a_\omega \cdot (z_{k,\omega} - r_{k,\omega}^{T_{k,\omega}})$, where $r_{k,\omega}^{T_{k,\omega}}$ was the mask used when $z_{k,\omega}$ was computed. Assume that $z_{k,\omega}$ was stored in the vth position of the block when it was computed. This implies that $r_{k,\omega}^{T_{k,\omega}} = \sum_{i|P_i \in T_{k,\omega}} \lambda_{i,v} \cdot r_{k,\omega,i}$. Now, using the sharing $[\![0\cdots0r_{k,\omega,i}0\cdots0]\!]_d$ the parties have for each $i \in [n]$, where $r_{k,\omega,i}$ is at the kth position, they locally compute:
     (a) $\left[\!\left[0\cdots0r_{k,\omega}^{T_{k,\omega}}0\cdots0\right]\!\right]_d = \sum_{i|P_i \in T_{k,\omega}} \lambda_{i,v} \cdot [\![0\cdots0r_{k,\omega,i}0\cdots0]\!]_d$.
     (b) $[\![0\cdots0r'_k0\cdots0]\!]_d = \sum_{\omega=1}^{w} a_\omega \cdot \left[\!\left[0\cdots0r_{k,\omega}^{T_{k,\omega}}0\cdots0\right]\!\right]_d$

     Upon computing the above for each $k \in [\ell]$, the parties locally compute

     $$[\![r'_1 \cdots r'_\ell]\!]_d = [\![r'_1 0\cdots0]\!]_d + [\![0r'_2 0\cdots0]\!]_d + \cdots + [\![0\cdots0r'_\ell]\!]_d .$$

     Finally, they locally compute $[\![x_1 \cdots x_\ell]\!]_d = [\![x'_1 \cdots x'_\ell]\!]_d + [\![r'_1 \cdots r'_\ell]\!]_d$. By repeating the above for both input blocks, the parties can proceed to compute the next multiplication operation as explained above.

Correctness. We show that if the parties act honestly, then they obtain the correct result. Namely, we prove that $\forall k \in [\ell] : \; x_k = \sum_{\omega=1}^{w} a_\omega \cdot z_{k,\omega}$.

This holds since in the above protocol $x_k = x'_k + r'_k$, where $x'_k = \sum_{\omega=1}^{w} a_\omega \cdot (z_{k,\omega} - r_{k,\omega}^{T_{k,\omega}})$ and $r'_k = \sum_{\omega=1}^{w} a_\omega \cdot r_{k,w}^{T_{k,\omega}}$. Observe that the correct $r_{k,w}^{T_{k,\omega}}$ is used. This follows from the fact that it was computed by taking the product of the shares of the parties in $T_{k,\omega}$ with the corresponding Lagrange coefficients.

Privacy. Exactly as in the single-secret-per-block case, each party uses an independent random mask for each multiplication operation. The only difference is that a block of several inputs is encoded over the same polynomial. Formally, privacy can be proved by the same argument as in Theorem 5.2 and so we omit the details.

Efficiency analysis. The communication required by our protocol to achieve privacy in the presence of malicious adversaries is the same as in the semi-honest protocol, with one difference: P_1 sends also the subset T that it used to compute the masked product. However, since this small data is sent *once for an entire layer of the program*, the increase in the cost is insignificant.

From a computational point of view, we now have a set of distributed keys for each party, and so computation grows by a factor of n. We stress that the vast amount of work, which is the conversion to Shamir's secret sharing can be done offline before the start of the actual computation as explained in Section 3.

5.2 Verifying Correctness of the Computation

In the previous section, we showed how to prevent leakage of private data during the computation of the program. However, nothing prevents a malicious adversary from cheating by sending false messages, causing the output to be incorrect. In this section, we add a step to our protocol, before the output is revealed, where the parties verify the correctness of the computation, and abort if cheating is detected. This additional step satisfies two desired properties: (i) it is a short constant-round protocol; (ii) it has *sublinear communication* in the size of the program, which means that *amortized* over the program, the communication cost remains the same.

The $\mathcal{F}_{\mathsf{vrfy}}$ ideal functionality. We begin by presenting the ideal functionality $\mathcal{F}_{\mathsf{vrfy}}$ to verify that multiplications were carried out correctly, and show how to compute it later. Recall that in our protocol, the parties hold at the beginning of each multiplication protocol, a degree-d sharing of ℓ left inputs and of ℓ right inputs. In addition, they hold a degree-d sharing of each of the program's outputs. $\mathcal{F}_{\mathsf{vrfy}}$ receives these from the honest parties, reconstruct the secrets and then check for each value, that it is correct *given* the values held by the parties as inputs for the multiplications that precede it. In addition, the honest parties send $\mathcal{F}_{\mathsf{vrfy}}$ also their shares of each input, as future shares depend on these values as well. We stress that it suffices for only the honest parties to send their shares, since they fully define the secrets (as we will see, a consistency check is carried out before calling $\mathcal{F}_{\mathsf{vrfy}}$ in our main protocol and so we are guaranteed at this stage that all sharings are consistent). The formal description appears in Functionality 5.3.

We note that $\mathcal{F}_{\mathsf{vrfy}}$ also uses the honest parties' shares to compute the corrupted parties' shares and hands them to the ideal world adversary $\mathcal{S}$. This is not problematic, since these are anyway known to the adversary in the main protocol which works in the $\mathcal{F}_{\mathsf{vrfy}}$-hybrid model. Also, $\mathcal{F}_{\mathsf{vrfy}}$ hands $\mathcal{S}$ the additive difference between the actual values and correct values. This is also an information already known to the adversary, as in DN-style multiplication protocols the adversary is even allowed to carry-out an additive attack over the output (see [50, 18, 31]).

FUNCTIONALITY 5.3 ($\mathcal{F}_{\mathsf{vrfy}}$- Verify Correctness of Multiplications)
$\mathcal{F}_{\mathsf{vrfy}}$ works with a set of honest parties and an ideal world adversary $\mathcal{S}$ who controls at most t parties. $\mathcal{F}_{\mathsf{vrfy}}$ receives as an input a layered SL program P with m bi-linear instructions. Then:

1. The honest parties sends $\mathcal{F}_{\mathsf{vrfy}}$ their shares of all P's inputs $\{[\![x_i \cdots x_i]\!]_d\}_{i=1}^{n}$, of all P's outputs $\{[\![o_i \cdots o_i]\!]_d\}_{i=1}^{n}$ and of all multiplication left and right input blocks $\left\{[\![u_{j,1} \cdots u_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$ and $\left\{[\![v_{j,1} \cdots v_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$.
2. $\mathcal{F}_{\mathsf{vrfy}}$ reconstructs all inputs x_i, outputs o_i and inputs to multiplication operations $u_1, \ldots, u_m, v_1, \ldots, v_m$. In addition, it computes the shares of the corrupted parties given the honest parties' shares, and sends them to $\mathcal{S}$.
3. $\mathcal{F}_{\mathsf{vrfy}}$ checks that each $z_\iota \in \{u_1, \ldots, u_m, v_1, \ldots, v_m, o_1, \ldots, o_n\}$ is correct given the preceding values, i.e., it checks for each $\iota \in [2m+n]$ that
$$\delta_\iota = z_\iota - \sum_{\omega=1}^{w} \alpha_\omega \cdot (\mu_{\iota,\omega} \cdot \nu_{\iota,\omega}) = 0$$
where $\mu_{\iota,\omega}, \nu_{\iota,\omega} \in \{1, x_1, \ldots, x_n, u_1, \ldots, u_m, v_1, \ldots, v_m\}$ according the structure of the program P. If not, it sends reject to the honest parties and $\mathcal{S}$. Otherwise, it sends accept to $\mathcal{S}$ and waits for $\mathcal{S}$ to send back $\mathsf{out} \in \{\mathsf{accept}, \mathsf{reject}\}$. Then, $\mathcal{F}_{\mathsf{vrfy}}$ sends out to the honest parties and halts. In addition, it sends δ_ι for each $\iota \in [2m+n]$ to $\mathcal{S}$.

The Boneh *et al.* [9] distributed verification protocol. We begin with describing the sub-linear verification protocol from [9] for Shamir sharings encoding a single secret. At the beginning of the protocol, the parties hold for each multiplication k, sharings $[\![x_k]\!]_d$ and $[\![y_k]\!]_d$ of the inputs and $[\![z_k]\!]_d$ of the output.

The idea is to define a verification circuit C which takes $\{x_k\}_{k=1}^m, \{y_k\}_{k=1}^m$ and $\{z_k\}_{k=1}^m$ as well as random public coefficients $\{\beta_k\}_{k=1}^m$, and outputs $\sum_{k=1}^m \beta_k \cdot (z_k - x_k \cdot y_k)$. Clearly, if $\forall k \in [m] : z_k - x_k \cdot y_k = 0$, then the output of C will be 0, whereas if $\exists k \in [m] : z_k - x_k \cdot y_k \neq 0$, then the output will be 0 only with probability $\frac{1}{|\mathbb{F}|}$. The main observation towards the protocol described next is that, the parties can locally compute a $2d$-degree sharing of the output of c. However, a degree-$2d$ sharing has no robustness and so we cannot let the parties simply open it and check equality to 0, since the corrupted parties could open it to any value they wish. Instead, the idea is to convert the $2d$-degree sharing into a d-degree sharing, verify that the conversion was done correctly, and then let the parties open the robustly shared output and check its equality to 0.

To this end, the parties split the circuit C into M identical g gates. Each g gate takes a random linear combination of $L = m/M$ multiplication triples, and then C takes a random linear combination of the g gates' outputs. Thus, the circuit C takes $3m$ inputs and each g gate takes $3L$ inputs (note that the random public coefficients can hardwired into the circuit)

Next, the parties define $3L$ polynomials $f_1, \ldots, f_{3L}$ as follows: $f_j(u)$ is the jth input to the uth g-gate. In addition, the parties generate an additional random g gate by taking random sharings as its input. Each of the $3L$ inputs to this gate is the free term of an f polynomial. Thus, each f polynomial is defined by $M+1$ points (since there are M copies of g gates in the circuit C and an additional random g gate). Next, the parties define another polynomial $Q(x)$ as $Q(x) = g(f_1(x), \ldots, f_{3L}(x))$. It follows from this definition, that for each $x \in \{0, \ldots, M\}$, $Q(x)$ is the output of the xth g-gate. Note however that $Q(x)$ has degree $2M$, since each f has degree M and the g gate is of degree 2. This means that the parties need to compute M additional points on Q in order to fully define it. This is done by computing shares of the inputs for M additional

g-gates via local interpolation and then having each party locally computing the $2d$-degree sharing of their outputs as above. At this point, the parties holds a d-degree sharings of $M+1$ points on the f polynomials, and a $2d$-degree sharing of $2M+1$ points on the Q polynomial. Next, the parties convert their shares of $2d$-degree polynomial into shares over a d-degree polynomial by secret sharing each share via a d-degree polynomial. In [9], the authors considered only the case where $n = 2d+1$, which means that the parties can simply sum the obtained shares to hold a secret sharing of degree d of $Q(1), \ldots, Q(M)$ (since their $2d$-degree shares can be locally converted to additive shares of the secret). In this work, we assume that $n > 2d+1$. The parties thus first convert the $2d$-degree sharings to additive sharings by locally adding $[\![0]\!]_{n-1}$, and then share their additive shares and continue as above. Then, they can take a linear combination of these shares and open the result to check equality to 0. Here the adversary cannot open to other values since the shares held by the honest parties uniquely determine the secret. However, this is not enough. A corrupted party can share an incorrect value and not its true additive share. In order to check that the parties shared the correct value, it suffices to check that Q is defined correctly. This is carried out by sampling a random point $r \in \mathbb{F} \setminus \{0, \ldots, M\}$ and checking that $Q(r) = g(f_1(r), \ldots, f_{3L}(r))$. To perform the check, the parties locally compute a d-degree sharing of $f_1(r), \ldots, f_{3L}(r)$ and $Q(r)$ via interpolation, open the obtained sharings, and check that $Q(r) = g(f_1(r), \ldots, f_{3L}(r))$. As before, the adversary can cause nothing beyond an abort, since all sharings are robustly shared using a d-degree polynomial.

The final crucial point is that by setting $M = L = \sqrt{m}$, we have that the overall communication cost in the above protocol is roughly $5n^2\sqrt{m} + 2n^2$ field elements (each party shares $2\sqrt{m}$ secrets and open $3\sqrt{m}+2$ sharings) which is sublinear in the size of the original program.

The cheating probability of the above protocol is at most $\frac{2M+1}{|\mathbb{F}|-M}$. This holds since if cheating took place, then the verification protocol can end successfully if one of two events occur: (1) the random linear combination yields an output of 0; (2) $Q(r) = g(f_1(r), \ldots, f_{3L}(r))$ for the sampled r even though $Q(x) \neq g(f_1(x), \ldots, f_{3L}(r))$. The first event happens with probability $\frac{1}{|\mathbb{F}|}$, whereas the second happens by the Schwartz-Zippel lemma with probability $\frac{2M}{|\mathbb{F}|-M}$. This implies that the overall cheating probability is bounded by $\frac{2M+1}{|\mathbb{F}|-M}$.

Adapting the verification protocol to packed secret sharing. In order to use the Boneh *et al.* verification method described above, observe first that the parties in our protocol do not hold a sharing of each multiplication's output. Rather, the parties hold a sharing of a linear combination of these outputs. Thus, the subcircuits for which the parties wish to verify equality to 0, are of the form $z_\iota - \sum_{j=1}^{J_\iota} \alpha_{\iota,j} \cdot (\mu_{\iota,j} \cdot \nu_{\iota,j}) = 0$ where z_ι is an input value for the ιth multiplication operation or an output value of the program, and should be the result of computing a linear combination of J_ι values (values for which $\alpha_{\iota,j} \neq 0$), and each such value is a result of multiplying two values $\mu_{\iota,j}$ and $\nu_{\iota,j}$. Note that the latter holds if the jth value in the summation came from a preceding multiplication. For the case that it is a program's input, we can simply let $\mu_{\iota,j}$ be the input and set $\nu_{\iota,j} = 1$. Note that there are m multiplications and n outputs, and so $\iota \in [2m+n]$ (since each z_ι is one of two inputs to a multiplication operation or an output value). Taking a random linear combination of the $m+n$ sub-circuits, we have that the parties wish to verify that

$$\sum_{\iota=1}^{2m+n} \beta_\iota \cdot \left(z_\iota - \sum_{j=1}^{J_\iota} \alpha_{\iota,j} \cdot (\mu_{\iota,j} \cdot \nu_{\iota,j}) \right) = 0 \tag{2}$$

where the random coefficients $\beta_1, \ldots, \beta_{m+n} \in \mathbb{F}$ will be chosen jointly by the parties at the beginning of the verification protocol by calling $\mathcal{F}_{\text{coin}}$.

Next, recall that each multiplication between two values can appear several times in Eq. (2). We thus can rewrite Eq. (2) as

$$\sum_{i=1}^{n} \beta_i \cdot o_i + \sum_{\iota=1}^{2m} \beta_\iota \cdot z_\iota - \sum_{j=1}^{m} \gamma_j \cdot (u_j \cdot v_j) - \sum_{i=1}^{n} \gamma_i \cdot x_i = 0 \tag{3}$$

where o_i is the ith output, for each $j \in [m]$: u_j and v_j are the left and right inputs for the jth multiplication, x_i is the ith input, and the coefficients γ_j and γ_i are obtained by summing the coefficients of the jth multiplication and the ith input respectively, from all its appearances in (2) (i.e., $\gamma_j = \sum_{\iota:\ \mu_{\iota,j}=u_j \wedge \nu_{\iota,j}=v_j} \beta_\iota \cdot \alpha_{\iota,j}$).

We thus have a circuit that takes $4m+2n$ inputs and outputs 0 if no cheating took place in the private computation of the program.

The next step would be to split the circuit into identical g gates and apply the above mechanism. However, this raises a problem. Recall that each input to the verification circuit is shared as part of a block of ℓ secrets. This means that each input is encoded at a different entry (i.e, a different point on the encoding polynomial). Now, when defining the f polynomials over the circuit's inputs, we need all the values that define a f polynomial to be encoded at the same entry, as otherwise interpolation to compute additional points will not be possible. Our solution to this is to organize the inputs such that the eth input to each g gate will be encoded at the same entry. We remark that for the program's inputs and outputs this is not an issue, as they are handled in a different way as explained below.

Let M be the number of g gates and let L be the number of inputs to each g gate. Setting $L = 4\ell\sqrt{m}$ and $M = \frac{\sqrt{m}}{\ell}$, we define a g gate as

$$g\left(\{\beta_\iota \cdot z_\iota\}_{\iota=1}^{2\ell\sqrt{m}}, \{(\gamma_j \cdot u_j, v_j)\}_{j=1}^{\ell\sqrt{m}}\right) = \sum_{\iota=1}^{2\ell\sqrt{m}} (\beta_\iota \cdot z_\iota) - \sum_{j=1}^{\ell\sqrt{m}} ((\gamma_j \cdot u_j) \cdot v_j). \tag{4}$$

Note that each gate takes L inputs. Note also that we consider the coefficients γ_j as *part of the input*, so that all g gates in our verification circuit will be identical. This is possible since multiplication with a public value is a local operation for our secret sharing scheme. Next, given that the size of the block is ℓ and since each g simply performs additions and linear product operations which are insensitive to the order of the inputs, we thus can reorder the inputs such that the ιth z_ι input is encoded in the $[\iota \bmod \ell]$ position of a block. Similarly, the jth multiplication pair of inputs are positions in the $[j \bmod \ell]$ entry of a block. By ordering the inputs for the g gates in this way, we obtain the property that the eth input to each g gate is positioned at the same entry and so the f polynomials are properly defined. Once the g gates are defined, we can apply the machinery of [9]. Recall that the parties need to locally compute an additive sharing of each g gate's output. For each multiplication operation between u_j and v_j, the parties locally multiply their shares, add a random sharing $[\![0 \cdots 0]\!]_{n-1}$ to the result, and then use Lagrange coefficients, corresponding to the position of $u_j \cdot v_j$ in the block, to convert it to an additive sharing of $u_j \cdot v_j$. For each z_ι, the parties will add a fresh new sharing $[\![0 \cdots 0]\!]_{n-1}$ to the shared block of where z_ι is encoded, and then convert it to an additive sharing of z_ι, using the appropriate Lagrange coefficients.

We note that the linear combination of the program's inputs and outputs is added at the end, after the parties verify that they hold a correct sharing of each g gate's output. This is possible

since we can view the sharings $[\![x_i \cdots x_i]\!]_d$ and $[\![o_i \cdots o_i]\!]_d$ simply as $[\![x_i]\!]_d$ and $[\![o_i]\!]_d$. Finally, when the parties have a d-degree sharing of (3), they can open the result and check equality to 0.

Formally, the verification protocol thus works as follows:

Π_{verify}: At the beginning of the protocol, the parties hold sharings of all P's inputs $\{[\![x_i \cdots x_i]\!]_d\}_{i=1}^n$, of all P's outputs $\{[\![o_i \cdots o_i]\!]_d\}_{i=1}^n$ and of all multiplication left and right input blocks $\left\{[\![u_{j,1} \cdots u_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$ and $\left\{[\![v_{j,1} \cdots v_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$.
From this point on, the parties view each sharing of an input and output value as $[\![x_i]\!]_d$ and $[\![o_i]\!]_d$ respectively. For each $y_\iota \in \{u_1, \ldots, u_m, v_1, \ldots v_m\}$, the parties view the sharing $[\![y_1 \cdots y_\ell]\!]_d$, where y_ι is encoded at the kth entry as $[\![y_\iota]\!]_d^{(k)}$ (i.e., a sharing of y_t via a d-degree polynomial, where y_ι is encoded at the point $-k+1$).
Let $L = 4\ell\sqrt{m}$ and $M = \frac{\sqrt{m}}{\ell}$. The parties work as follows:

- **Pre-processing**: The parties call $\mathcal{F}_{\text{LinRand}}$ to receive $4m$ sharings of the form $[\![0 \cdots 0]\!]_{n-1}$ and L sharings of the form $[\![r_1 \cdots r_\ell]\!]_d$.
- **Round 1:**
  1. The parties call $\mathcal{F}_{\text{coin}}$ to obtain random $\beta_1, \ldots, \beta_{2m+n} \in \mathcal{F}$.
  2. The parties locally compute for each $j \in [m]$ and $i \in [n]$ the coefficients γ_j and γ_i in Eq. (3) as explained in the text, according to the program structure. Then, $\forall \iota \in [2m]$: set $z'_\iota = \beta_\iota \cdot z_\iota$ and $\forall j \in [m]$: set $u'_j = \gamma_j \cdot u_j$.
  3. The parties define M copies of a g gate as in Eq. (4). The parties order the inputs such that the eth input to all g gate are encoded at the same position k as explained in the text.
  4. The parties define L polynomials $f_1, \ldots, f_L \in \mathbb{F}[x]$ of degree M such that $\forall e \in [L]$: (i) $\forall h \in [M]$, $f_e(h)$ is the eth input to hth g gate; (ii) $f_e(0)$ is a random element. For (ii), the parties utilize the next unused $[\![r_1 \cdots r_\ell]\!]_d$. If the points defined for f_e are encoded at the kth entry, then the parties interpret $[\![r_1 \cdots r_\ell]\!]_d$ as $[\![r_k]\!]_d^{(k)}$.
  5. For each $e \in [L]$: given $[\![f_e(0)]\!]_d^{(k_e)}, \ldots, [\![f_e(M)]\!]_d^{(k_e)}$, the parties locally compute $[\![f_e(M+1)]\!]_d^{(k_e)}, \ldots, [\![f_e(2M)]\!]_d^{(k_e)}$ via Lagrange interpolation.
  6. Let $p(x) = g(f_1(x), \ldots, f_L(x))$ a $2M$-degree polynomial. The parties locally compute additive shares of $p(0), \ldots, p(2M)$ by computing Eq. (4) for each $p(h)$ with $h \in \{0, \ldots, 2M\}$ in the following way:
     (a) For each $\iota \in [2\ell\sqrt{m}]$: compute $[\![z'_\iota]\!]_d^{(k_\iota)} + [\![0 \cdots 0]\!]_{n-1}$ and convert it to an additive sharing of z'_ι using Lagrange coefficients. Denote the share held by P_i by $z'_{\iota,i}$
     (b) For each $j \in [\ell\sqrt{m}]$, the parties locally multiply $\left[\!\left[u'_j\right]\!\right]_d^{(k_j)} \cdot [\![v_j]\!]_d^{(k_j)} + [\![0 \cdots 0]\!]_{n-1}$ and convert the result to an additive sharing of $u'_j \cdot v_j$ using Lagrange coefficients. Denote the share held by P_i by $(u'_j \cdot v_j)_i$.
     (c) Each party P_i locally computes $\sum_{t=1}^{2\ell\sqrt{m}} z'_{\iota,i} - \sum_{j=1}^{\ell\sqrt{m}} (u'_j \cdot v_j)_i$.

7. Denote the shares of $p(0), \ldots, p(2M)$ held by party P_i by $p(0)^i, \ldots, p(2M)^i$, Then, each party P_i generates sharings $[\![p(0)^i]\!]_d, \ldots, [\![p(2M)^i]\!]_d$ (these are standard Shamir sharings of degree d where the secret is encoded at the point 0) and distributes the shares to the other parties.

- **Round 2:** The parties run a consistency check to all sharings receives in the previous step. If it fails, then the parties abort. Otherwise, each party sums its shares to obtain $[\![p(0)]\!]_d, \ldots, [\![p(2M)]\!]_d$.

- **Round 3:**

1. The parties call $\mathcal{F}_{\text{coin}}$ to sample $r \in \mathbb{F} \setminus \{0, \ldots, M\}$.
2. The parties locally compute $[\![f_1(r)]\!]_d^{(k_1)}, \ldots, [\![f_L(r)]\!]_d^{(k_L)}$ and $[\![p(r)]\!]_d$ using Lagrange interpolation.
3. The parties open these sharings by sending the shares to each other. If there exists a party for which the opening fails due to inconsistency, then this party sends abort to the other parties and aborts. Otherwise, each party checks that $p(r) = g(f_1(r), \ldots, f_L(r))$. If the equality holds, then the parties proceed to the next step. Otherwise, they output reject and halt.

- **Round 4:**

1. The parties locally compute $[\![b]\!]_d = \sum_{h=1}^{M} [\![p(h)]\!]_d + \sum_{i=1}^{n} \beta_i \cdot [\![o_i]\!]_d - \sum_{i=1}^{n} \gamma_i \cdot [\![x_i]\!]_d$.
2. The parties open $[\![b]\!]_d$ by sending their shares to each other. If any party receives inconsistent shares, then it sends abort to the other parties and aborts. Otherwise, each party checks that $b = 0$. If this does not hold, then the parties output reject. Otherwise, the parties output accept.

Communication cost. For sharing its additive shares of $p(0), \ldots, p(2M)$, each party sends $(n-1) \cdot (2M+1)$ field elements. For opening $f_1(r), \ldots, f_L(r), p(r)$ and b, each party sends $(n-1) \cdot (L+2)$ elements. Since $L = 4\ell\sqrt{m}$ and $M = \frac{\sqrt{m}}{\ell}$ the overall communication cost per party is $(n-1) \cdot (2M + L + 3) = (n-1) \cdot (2\frac{\sqrt{m}}{\ell} + 4\ell\sqrt{m} + 3)$ field elements, which is sublinear in the size of the program m (note that ℓ will typically be a small constant in instantiations of our protocol).

Security. We next prove that our protocol realizes the $\mathcal{F}_{\mathsf{vrfy}}$ ideal functionality.

Lemma 5.4 *Protocol Π_{verify} securely computes $\mathcal{F}_{\mathsf{vrfy}}$ with abort and with statistical error $\frac{2M+1}{|\mathbb{F}|-M}$ in the $(\mathcal{F}_{\text{LinRand}}, \mathcal{F}_{\text{coin}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

Proof: Let $\mathcal{S}$ be the ideal world adversary and let $\mathcal{A}$ be the real world adversary. $\mathcal{S}$ is invoked by $\mathcal{F}_{\mathsf{vrfy}}$ which sends it all the corrupted parties' shares in the sharings of P's inputs $\{[\![x_i \cdots x_i]\!]_d\}_{i=1}^n$, of P's outputs $\{[\![o_i \cdots o_i]\!]_d\}_{i=1}^n$ and of multiplications left and right input blocks $\left\{[\![u_{j,1} \cdots u_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$ and $\left\{[\![v_{j,1} \cdots v_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$. In addition, $\mathcal{F}_{\mathsf{vrfy}}$ sends $\mathcal{S}$ the additive differences d_ι for each $\iota \in [2m+n]$.

In the simulation, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{coin}}$, thus choosing and handing $\mathcal{A}$ random $\beta_1, \ldots, \beta_{2m+n} \in \mathbb{F}$ and $r \in \mathbb{F} \setminus \{0, \ldots, M\}$. In addition, by playing the role of $\mathcal{F}_{\text{LinRand}}$ in the pre-processing, the simulator $\mathcal{S}$ knows all shares held by the corrupted parties of any random sharing (this holds

by the definition of $\mathcal{F}_{\text{LinRand}}$). Note that this (together with the shares received from $\mathcal{F}_{\mathsf{vrfy}}$) allows $\mathcal{S}$ to compute all the additive shares $p(0)^i, \ldots, p(2M)^i$ for each corrupted party P_i and so it knows at the end of the first round whether $\mathcal{A}$ shared the correct values or not ($\mathcal{S}$ can reconstruct it from the honest parties' shares sent to him by $\mathcal{A}$ and compare it to what should have been sent). For each $j \in \{0, \ldots, 2M\}$, define $\Delta_j = \sum_{i:\ P_i \text{ corrupt}} (\alpha^i - p(j)^i)$, where α^i is the actual secret that was shared by each corrupted party P_i. Thus, defining the $2M$-degree polynomial $q(x) = p(x) - g(f_1(x), \ldots, f_L(x))$, we have that $\forall j \in \{1, \ldots, 2M\} : \ q(j) = \Delta_j$ and so $\mathcal{S}$ can compute any points it wishes on q. In addition, since $\mathcal{S}$ knows all the d_ι values and $\Delta_1, \ldots, \Delta_M$, it can compute b (the output of the verification circuit) by taking $b = \sum_{\iota=1}^{2m+n} \beta_\iota \cdot d_\iota + \sum_{j=1}^{M} \Delta_j$.

Based on the above, $\mathcal{S}$ works as follows:

- For each honest party P_i, the simulator $\mathcal{S}$ sends t random shares for $p(0)^i, \ldots, p(2M)^i$ to $\mathcal{A}$. Since t shares can open to any value, this messages are distributed the same as the in real world execution.

- For simulating the opening of b, the simulator $\mathcal{S}$ computes the corrupted parties' shares of b and then chooses random shares to the honest parties, given the corrupted parties' shares and under the constraint that they will open to b (computed as above). Observe that the simulation in this step is in fact perfect.

- For simulating the opening of $f_1(r), \ldots, f_L(r)$, the simulator $\mathcal{S}$ computes the corrupted parties' shares, chooses random elements for these values and chooses random shares for the honest parties' given the chosen values and the corrupted parites's shares. Note that since the constant term of each polynomial is completely random, then the distribution is again the same as in the real world execution.

- Finally, $\mathcal{S}$ needs to simulate the opening of $p(r)$. For this, $\mathcal{S}$ computes $g(f_1(r), \ldots, f_L(r))$ and $q(r)$ and set $p(r) = q(r) + g(f_1(r), \ldots, f_L(r))$. Then, it computes the corrupted parties' shares and chooses the honest parties' shares, given these shares, and under the constraint that they will open to $p(r)$. Note that $p(r)$ in the simulation is random under the constraint that $q(r) = p(r) - g(f_1(r), \ldots, f_L(r))$, exactly as in the real world execution.

- If $\mathcal{A}$ sends inconsistent shares to an honest party P_j, then $\mathcal{S}$ sends abort_j to $\mathcal{F}_{\mathsf{vrfy}}$. Otherwise, if or one of the two tests in the protocol did not pass, then $\mathcal{S}$ sends reject to $\mathcal{F}_{\mathsf{vrfy}}$. If $\mathsf{out} = \mathsf{reject}$ and the two tests passed successfully (meaning that the honest parties in the simulation output accept), then $\mathcal{S}$ outputs fail and halts. Otherwise, it outputs whatever $\mathcal{A}$ outputs.

Observe that the only difference between the simulation and the real world execution is the event that $\mathcal{S}$ outputs fail. This event happens when $\mathcal{S}$ receives reject from $\mathcal{F}_{\mathsf{vrfy}}$ (meaning that $\exists \iota : \delta_\iota \neq 0$) but the honest parties in the simulation outputs accept since $b = 0$ and $p(r) = g(f_1(r), \ldots, f_L(r))$. The former can happen with probability $\frac{1}{|\mathbb{F}|}$ (the random coefficients cause the additive differences to cancel each other) whereas the latter happens if $q(r) = 0$, i.e., with probability $\frac{2M}{|\mathbb{F}|-M}$ (by the Schwartz-Zippel lemma, since q is of degree M and r is chosen from $\mathbb{F} \setminus \{1, \ldots, M\}$). Thus, $\Pr[\mathsf{fail}]$ is identical to the probability of the event where the honest parties output accept in the real execution when cheating took place. Thus, the statistical error is exactly as allowed by the theorem. This concludes the proof. ■

5.3 Putting It All Together - The Main Protocol

We are now ready the present our main protocol with security against malicious adversaries. The protocol works by having the parties run the private protocol to compute the program, and then, before revealing the output, call the ideal functionality $\mathcal{F}_{\mathsf{vrfy}}$ to verify that the sharings they obtained throughout the execution, are correct. Since $\mathcal{F}_{\mathsf{vrfy}}$ requires the sharings it receives to be consistent, then the parties run a batch consistency check before calling $\mathcal{F}_{\mathsf{vrfy}}$.

Stragglers Resilience. We now show what resilience our protocol guarantees:

- *Input sharing step*: In this step, we require the parties to send a masked input $\hat{x}_i = x_i + r$ to all parties and not only to P_1. Looking on an epoch that consists of parties sending their masked input to the other parties, and then sending messages to P_1 in the first layer of bi-linear instructions, it is easy to see that even if $n - (2d + 1)$ messages are lost, party P_1 will receive $2d$ messages and will be able to proceed to the next epoch.

- *Private computation of the program*: Our new protocol in Section 5.1 can handle $n - (2d + 1)$ dropped messages in each epoch.

- *Verification step*: A subtle issue that arises here is the effect of stragglers existence in the private protocol, on the consistency check and $\mathcal{F}_{\mathsf{vrfy}}$. Specifically, if different subset of parties participate in each epoch, then the sharings used in the consistency check and $\mathcal{F}_{\mathsf{vrfy}}$ are held by different subset of parties, which seems problematic. Nevertheless, we observe that the number of such subsets is bounded by the depth of the program. Hence, we have three possible solutions. If the depth of the program is low, then the parties can run these two steps for each subset separately (recall that each such subset is of size $2d+1$ and so an honest majority required by the protocols is guaranteed). Since the cost in these final steps is anyway low and sublinear in the size of the program, we can afford running them several times. If the depth is larger than the number of possible subsets $\binom{n}{\tau}$ (with τ being the number of stragglers), then we can simply go over all possible subsets. Alternatively, if the program is very deep, then one can simply assume that all messages that were delayed during the computation, arrive by the time the parties reach the final steps. While this seems as a slight weakening of our stragglers-resilience model, note that even with this assumption, our protocol has a huge advantage over protocols with no resilience to stragglers, where the parties need to wait for all messages to arrive when computing *each layer*, and not only at the end of the entire computation.

  Note that in the former solution we need to assume that no messages are dropped inside this step, since in each subset of $2d+1$ parties, if a message is lost, we might lose the honest majority and hence the security guarantees. Since this step is a short constant-round protocol, this seems as a mild assumption.

- *Output Reconstruction*: If $2d+1$ shares arrive to each party, then at least $d+1$ shares are sent by honest parties and so are correct. This implies that the party can either reconstruct its correct output or abort if cheating took place. Thus, this step can also withstand $n-(2d+1)$ stragglers.

The formal description appears in Protocol 5.6. We thus obtain a maliciously-secured protocol, with the same (amortized) communication cost and same stragglers resilience as for semi-honest adversaries (with a small caveat for the short verification step). This is summarized in the following Theorem:

Theorem 5.5 *Let $\mathbb{F}$ be a finite field, let f be a n-party functionality represented by a ℓ-layered straight-line program over $\mathbb{F}$ with S bilinear instructions, let t be a security threshold parameter and let d be a parameter such that $d \geq t + \ell - 1$, $n \geq 2d + 1$ and $|\mathbb{F}| > n + d + \ell + 1$. Then, Protocol 5.6 computes f in the $(\mathcal{F}_{\text{LinRand}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\mathsf{vrfy}})$-hybrid model with t-malicious-security-with-abort, $(n - (2d + 1))$-stragglers-resilience, with statistical error $\frac{1}{|\mathbb{F}|}$, and communication cost of $\left(\frac{3}{\ell} - \frac{2d+3}{n \cdot \ell}\right) S + o(S)$ field elements sent per party.*

The protocol has statistical error of $\frac{1}{|\mathbb{F}|}$ due to the consistency check that may fail. As explained above, for small fields the error can be reduced by repeating the check with independent randomness.

Proof: We construct an ideal world simulator $\mathcal{S}$ that interacts with a real world adversary $\mathcal{A}$. In the execution, $\mathcal{S}$ will play the role of the trusted party computing the ideal functionalities $\mathcal{F}_{\text{LinRand}}$ and $\mathcal{F}_{\text{coin}}$ and the honest parties. Note that by playing the role of $\mathcal{F}_{\text{LinRand}}$, $\mathcal{S}$ *knows all the randomness being used throughout the execution.* The simulation works as follows:

- *Input sharing step*: For the honest parties' inputs, $\mathcal{S}$ uses the input '0' and simulates honest behavior. Once receiving $x_i + r$ from $\mathcal{A}$ for each corrupted party P_i, the simulator $\mathcal{S}$ uses its knowledge of r to extract the corrupted parties' shares. If $\mathcal{A}$ sent different values to the honest parties or if it causes an abort in the comparison of views, then $\mathcal{S}$ simulates the honest parties aborting the protocol, sends abort to the trusted party computing f and outputs whatever $\mathcal{A}$ outputs. A crucial point here is that $\mathcal{S}$ knows at the end of this step not only the inputs themselves but also all shares held by $\mathcal{A}$, since it knows the random shares every party holds.

- *Computing the program*: The simulator plays the role of the honest parties. If it plays the role of P_1, then it receives the masked shares sent by the corrupted parties to P_1. Since it knows the corrupted parties' shares of the inputs to each multiplication and its randomness, it knows exactly whether $\mathcal{A}$ cheats or not. If P_1 is corrupted, then $\mathcal{S}$ computes in his head the shares that should be sent by P_1 to the honest parties, and uses them to detect cheating. By Theorem 5.2, we have that the view of the adversary in this step is distributed identically to its view in the real execution.

- *Consistency Check*: $\mathcal{S}$ plays the role of the $\mathcal{F}_{\text{coin}}$ handing all αs to $\mathcal{A}$ and the role of the honest parties. If the execution ends with the honest parties aborting, then $\mathcal{S}$ sends abort to the trusted party computing f, outputs whatever $\mathcal{A}$ outputs and halts. If the shares that were dealt by a corrupted party were not consistent, but the honest parties did not detect it in the execution of the check, then $\mathcal{S}$ outputs fail and halts. Otherwise, the simulation proceeds to the next step.

- *Verification of multiplications*: In this step, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{vrfy}}$. If cheating took place during the emulation of the computation, then $\mathcal{S}$ sends reject to $\mathcal{A}$, simulates the honest parties aborting in the real execution, sends abort to the trusted party computing f and outputs whatever $\mathcal{A}$ outputs. The same applies for the case where no cheating took place, $\mathcal{S}$ sends accept to $\mathcal{A}$, but $\mathcal{A}$ decides to change the output to reject (which is allowed by the definition of $\mathcal{F}_{\mathsf{vrfy}}$). Otherwise, $\mathcal{S}$ proceeds to the next step.

- *Outputs*: $\mathcal{S}$ receives the output of the corrupted parties' from the trusted party computing f. Then, it chooses new shares for the honest parties, given the output and the shares held by the corrupted parties. Then, it sends them to $\mathcal{A}$. For each output intended to a honest party P_j, if $\mathcal{A}$ sent inconsistent shares, then it sends abort_j to the trusted party. Otherwise, $\mathcal{S}$ sends $\mathsf{continue}_j$ to the trusted party. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Observe that when the event that $\mathcal{S}$ outputs fail does not occur, then the only difference between the simulation and the real execution is in the input sharing step and the output reconstruction. However, by the privacy of the secret sharing scheme, the shares $\mathcal{A}$ sees are distributed identically in both executions. Next, as explained in the text, $\mathcal{S}$ outputs fail with probability $\frac{1}{|\mathbb{F}|}$ (this is the probability that a random combination of inconsistent sharings will be consistent). This is exactly the statistical error allowed by the theorem. It follows that the only difference between the simulation and the real execution is the event that $\mathcal{S}$ outputs fail which happens with probability $\frac{1}{|\mathbb{F}|}$. This concludes the proof. ■

PROTOCOL 5.6 (Computing a layered SLP with Malicious Security)

The parties $P_1, \ldots, P_n$ hold a description of a layered straight-line program over $\mathbb{F}$, with m bilinear instructions partitioned to batches of ℓ instructions, such that the inputs to each batch depends only on previous batches. Let ϕ_P be the repetition pattern induced by P.

- **Pre-processing**: The parties call $\mathcal{F}_{\text{LinRand}}$ to obtain random sharings $[\![r \cdots r]\!]_d$ for each input/output of P, and to obtain for the jth bilinear instruction sharings $[\![r_{j,i}0 \cdots 0]\!]_d, \ldots, [\![0 \cdots 0 r_{j,i}]\!]_d$, where $r_{j,i}$ is known to P_i, for each $\imath \in [n]$.
- **The Protocol**:

  1. The parties emulate the program's instructions as follows:
     (a) *Load an input to memory*: For each instruction $R_j \leftarrow x_i$, with x_i being held by party P_i and $[\![r \cdots r]\!]_d$ being the random sharing that was assigned to the ith input:
        i. The parties send P_i their shares of $[\![r \cdots r]\!]_d$.
        ii. If the shares are inconsistent, then party P_i sends abort to all the other parties and outputs $\perp$. Otherwise, it reconstructs r and sends $\hat{x}_i = x_i + r$ to all parties.
        iii. The parties compare their view of $\hat{x}_i$. If any party received a different $\hat{x}_i$ from the one it received from P_i, then it sends abort to all the other parties, outputs $\perp$ and halts.
        Remark: Note that this check can be performed with small constant cost for all inputs together by using a collision-resistant hash function and having each party sending a hash of all masked inputs or by taking a random linear combination of the masked inputs.
        iv. The parties locally compute $[\![x_i \cdots x_i]\!]_d = \hat{x}_i - [\![r \cdots r]\!]_d$.
     (b) *Evaluating the jth batch of "multiply two linear combinations" instructions*: Run the private protocol described in Section 5.1.
  2. *Consistency check*: Let $\left\{[\![v_{j,1} \cdots v_{j,\ell}]\!]_d\right\}_{j=1}^{2m}$ all sharings dealt by P_1 during the execution. The parties work as follows:
     (a) The parties call $\mathcal{F}_{\text{coin}}$ to receive $\alpha_1, \ldots, \alpha_{2m} \in \mathbb{F}$ and call $\mathcal{F}_{\text{LinRand}}$ to receive $[\![r_1 \cdots r_\ell]\!]_d$.
     (b) The parties locally compute $[\![z_1 \cdots z_\ell]\!]_d = \sum_{j=1}^{2m} \alpha_j \cdot [\![v_{j,1} \cdots v_{j,\ell}]\!]_d + [\![r_1 \cdots r_\ell]\!]_d$.
     (c) The parties open $[\![z_1 \cdots z_\ell]\!]_d$ by sending their shares to all the other parties. If any party received inconsistent shares, then it sends abort to all parties, outputs $\perp$ and halts.
  3. *Verifying correctness*: The party call $\mathcal{F}_{\mathsf{vrfy}}$ by sending their shares of all inputs $\{[\![x_i \cdots x_i]\!]_d\}_{i=1}^{n}$, of all outputs $\{[\![o_i \cdots o_i]\!]_d\}_{i=1}^{n}$ and of all multiplication left input blocks $\left\{[\![u_{j,1} \cdots u_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$ and right input blocks $\left\{[\![v_{j,1} \cdots v_{j,\ell}]\!]_d\right\}_{j=1}^{m/\ell}$.
     If $\mathcal{F}_{\mathsf{vrfy}}$ sends reject, then the parties output $\perp$ and halt. Otherwise, they proceed to the next step.
  4. If any party received abort in any of the previous steps, then it outputs $\perp$ and halts.
  5. *Emulating "output value from memory" instructions:* for each instruction $O_i \leftarrow R_j$, where P_i should receive O_i, the parties send their shares of the value in R_j to P_i. If the shares are inconsistent, then P_i sends abort to the other parties, outputs $\perp$ and halts. Otherwise, P_i reconstructs and outputs O_i.

Acknowledgements

We thank Tuvi Etzion for helpful pointers to the literature on covering designs.

E. Boyle supported by ISF grant 1861/16, AFOSR Award FA9550-21-1-0046, a Google Research Award, and ERC Project HSS (852952). N. Gilboa supported by ISF grant 2951/20, ERC grant 876110, and a grant by the BGU Cyber Center. Y. Ishai supported by ERC Project NTSC (742754), NSF-BSF grant 2015782, BSF grant 2018393, and ISF grant 2774/20. A. Nof supported by ERC Project NTSC (742754).

References

[1] Covering Designs. https://www.dmgordon.org/cover//.

[2] N. Alon, M. Merritt, O. Reingold, G. Taubenfeld, and R.N. Wright. Tight bounds for shared memory systems accessed by byzantine processes. *Distributed Computing*, 2005.

[3] S. Badrinarayanan, A. Jain, N. Manohar, and A. Sahai. Secure MPC: laziness leads to GOD. In *ASIACRYPT*, 2020.

[4] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In *ACM PODC*, 2014.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *ACM STOC*, 1988.

[6] R. Bendlin and I. Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In *TCC*, 2010.

[7] F. Benhamouda, E. Boyle, N. Gilboa, S. Halevi, Y. Ishai, and A. Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In *TCC*, 2021.

[8] K. A. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM CCS*, 2017.

[9] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, 2019. Full version: ePrint report 2019/188.

[10] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO*, 2019.

[11] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Correlated pseudorandom functions from variable-density LPN. In *FOCS*, 2020.

[12] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *ACM CCS*, 2019.

[13] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *ASIACRYPT*, 2020.

[14] Z. Brakerski, N. Chandran, V. Goyal, A. Jain, A. Sahai, and G. Segev. Hierarchical functional encryption. In *ITCS*, 2017.

[15] R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[16] R. Canetti and S. Goldwasser. An efficient *Threshold* public key cryptosystem secure against adaptive chosen ciphertext attack. In *EUROCRYPT*, 1999.

[17] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *ACM STOC*, 1988.

[18] 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 *CRYPTO*, 2018.

[19] A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk. Fluid MPC: secure multiparty computation with dynamic participants. In *CRYPTO*, 2021.

[20] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *USENIX*, 2017.

[21] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC*, 2005.

[22] I. Damgård and Y. Ishai. Scalable secure multiparty computation. In *CRYPTO*, 2006.

[23] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, 2010.

[24] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. D. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, 2008.

[25] I. Damgård and J. Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

[26] I. Damgård and R. Thorbek. Non-interactive proofs for integer multiplication. In *EUROCRYPT*, 2007.

[27] M. K. Franklin and M. Yung. Communication complexity of secure computation (extended abstract). In *ACM STOC*, 1992.

[28] Z. Füredi. Turán type problems. *Surveys in combinatorics*, 166:253–300, 1991.

[29] J. Furukawa and Y. Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In *ACM CCS*, 2019.

[30] G.Beck, A. Goel, A. Jain, and G. Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. In *EUROCRYPT*, 2021.

[31] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. Circuits resilient to additive attacks with applications to secure computation. In *ACM STOC*, 2014.

[32] N. Gilboa and Y. Ishai. Compressing cryptographic resources. In *CRYPTO*, 1999.

[33] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[34] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *ACM STOC*, 1987.

[35] S. Goldwasser and Y. Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18, 2005.

[36] D. Gordon, S. Ranellucci, and X. Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT*, 2018.

[37] D. M. Gordon and D. R. Stinson. Coverings. In *Handbook of Combinatorial Designs*, pages 391–398. 2006.

[38] S. D. Gordon, D. Starin, and A. Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In *EUROCRYPT*, 2021.

[39] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song. ATLAS: efficient and scalable MPC in the honest majority setting. In *CRYPTO*, 2021.

[40] V. Goyal, Y. Liu, and Y. Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *CRYPTO*, 2019.

[41] V. Goyal, A. Polychroniadou, and Y. Song. Unconditional communication-efficient MPC via hall's marriage theorem. In *CRYPTO*, 2021.

[42] V. Goyal, Y. Song, and C. Zhu. Guaranteed output delivery comes free in honest majority MPC. In *CRYPTO*, 2020.

[43] Y. Guo, R. Pass, and E. Shi. Synchronous, with a chance of partition tolerance. In *CRYPTO*, 2019.

[44] Huaxiong H. Wang and J. Pieprzyk. Shared generation of pseudo-random functions with cumulative maps. In *CT-RSA*, 2003.

[45] V. Hadzilacos. Issues of fault tolerance in concurrent computations (databases, reliability, transactions, agreement protocols, distributed computing). *PhD thesis*, 1985.

[46] S. Halevi, Y. Ishai, E. Kushilevitz, and T. Rabin. Best possible information-theoretic MPC. In *TCC*, 2018.

[47] M. Hirt and M. Mularczyk. Efficient MPC with a mixed adversary. In *Information-Theoretic Cryptography ITC*, 2020.

[48] I. Keidar and A. Shraer. How to choose a timing model. *IEEE Trans. Parallel Distrib. Syst.*, 19, 2008.

[49] C. Y. Koo. Secure computation with partial message loss. In *TCC*, 2006.

[50] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, 2017.

[51] S. Micali and R. Sidney. A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems. In *CRYPTO*, 1995.

[52] P. Raipin Parvédy and M. Raynal. Uniform agreement despite process omission failures. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[53] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12, 1986.

[54] M. Raynal. Consensus in synchronous systems: A concise guided tour. In *Symposium on Dependable Computing (PRDC)*, 2002.

[55] A. Shamir. How to share a secret. *Commun. ACM*, 1979.

[56] A. Sidorenko. What we know and what we do not know about turán numbers. *Graphs and Combinatorics*, 11(2):179–199, 1995.

[57] P. Turán. On an external problem in graph theory. *Mat. Fiz. Lapok*, 48:436–452, 1941.

[58] P. Turán. Research problems. *Közl MTA Mat. Kutató Int*, 6:417–423, 1961.

[59] A. Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[60] V. Zikas, S. Hauser, and U. Maurer. Realistic failures in secure multi-party computation. In *TCC*, 2009.

A Other Applications of Generalized PRSS

While our new PRSS results are presented in the context of general MPC, they can be used to obtain similar resilience-efficiency tradeoffs for a variety of other cryptographic applications. One class of such applications discussed by Cramer et al. [21] is to threshold cryptosystems. Here we briefly discuss simpler applications related to distributed storage or secure aggregation.

Consider the task of secure distributed storage. For instance, suppose that a client wants to distribute a password or digital currency s between multiple cloud servers by communicating a share s_i to each server. A standard secret-sharing scheme requires the shares s_i to be distinct. Using PRSS, following a one-time setup phase in which replicated keys are distributed to the servers, the client can just send to all servers the *same* masked secret $s + r$, where r is a pseudorandom secret which is locally computed from the PRSS keys. The servers can then convert their local Shamir-shares of r into local Shamir-shares of s by adding the mask $s + r$. Since sending the same message is typically much cheaper than sending multiple messages (e.g., using multicast protocols, gossip, or simply by posting on a social network), this PRSS-based solution is more practical.

The above distributed storage application can be easily extended to simple MPC applications that use linear secret sharing for secure summation or aggregation; see, e.g., [20, 8, 46] for some use cases. In such applications, clients share their secret inputs between the servers, and the servers aggregate the client values by summing up the shares received from the clients.

Unlike our main motivating application of MPC with strong honest majority, the above applications of PRSS are meaningful even when the polynomial degree satisfies $n/2 \leq d < n$. When $d = n - 2$ this is isomorphic to additive shares of 0, for which an optimal solution for arbitrary t was given in [32]. Our generalized PRSS construction is relevant to the case where $d < n-2$, which is useful in the context of the above applications for adding robustness or straggler resilience.

Micali and Sidney [51] use a similar kind of cover designs for reducing the amount of replicated keys compared to the baseline $\binom{n}{t}$ solution. Our generalized PRSS approach has the advantage of *compression*: following the one-time PRSS setup, the servers only need to store or communicate a compact Shamir-shares of each secret. This comes at a minor increase in the number of seeds induced by our transformation from (n, m, t)-cover to PRSS of degree-$(m-1)$ polynomials.

B Generating Double Shamir Sharing - A Second Approach

In this section, we present an alternative to the construction of Section 3.5, which reduces the number of seeds (by a factor of less than 2) at the cost of increasing the number of calls to a PRF. Recall that the goal is to generate two random polynomials of degree d and $2d$ which store the same ℓ random secrets.

Theorem B.1 *Fix integers $d > t > 0$ and $n > 2d$ and let $\ell = d - t + 1$. A size-k' $(n, d+1, t)$-cover can be used to construct a solution for t-secure distribution of double-Sharing of degree-d and degree-$2d$ polynomials, both packing the same ℓ elements, with the following complexity measures:*

- *The number of distinct subsets (seeds) is at most $k \leq k'(d+1)$.*
- *The total subset size (storage) is $\sum_j |S_j| \leq k'((d+1)(n-d))$.*
- *The total number of PRF calls is $k'(d+1)((n-d) + (n-d)(d-\ell+1))$.*

Proof: We proceed to the alternative construction. Let $\mathcal{C}' = (S'_1, \ldots, S'_{k'})$ be a size-k' $(n, d+1, t)$-cover. As in the previous section, we consider the set $\bar{\mathcal{C}}$ which contains all the subsets that are obtained by removing one element from any of the S'_j's. There are at most $k \leq k'(d+1)$ distinct subsets in $\bar{\mathcal{C}}$, each of size d. Denote the subsets in $\bar{\mathcal{C}}$ by $\bar{S}_1, \bar{S}_2, \ldots, \bar{S}_k$. We generate the polynomial $P_1(x)$ from these subsets exactly as in the Theorem 3.3. Next, we show how to generate $R(x)$ from these subsets. For each $\gamma \in \{-d+\ell, \ldots, 0\}$, let $R_{\gamma,\bar{S}_j}$ be the unique polynomial of degree $2d - \ell$ interpolated from

$$R_{\gamma,\bar{S}_j}(X) = \begin{cases} 0 & \text{if } X \in \bar{S}_j \\ 0 & \text{if } X \in \{-d+\ell, \ldots, 0\} \setminus \{\gamma\} \\ 1 & \text{if } X = \gamma \end{cases}$$

Now, letting $R(x) = \sum_j \sum_\gamma x_{\gamma,j} \cdot R_{\gamma,\bar{S}_j}$, where $x_{k,\gamma}$ is a random element given to all parties that are *not* in $\bar{S}_j$, yields a random polynomial which is well-defined (i.e., the parties have enough information to compute it) and of degree $2d - \ell$ as required.

Let M be the matrix defined by $M[i, (\gamma, j)] = R_{\gamma,\bar{S}_j}(i)$. To prove security against a subset of t parties, we need to show that for every subset T of size t, the submatrix $M'_{\bar{T}}$ has rank at least $2d - \ell - t + 1$. To see this, we fix a subset $T \subset [n]$ of size t, and denote the subset in $\mathcal{C}'$ that covers it by S'. Consider the submatrix $M_{\bar{T},S'}$ of $M'_{\bar{T}}$ consisting of columns for $[n] \setminus S' \cup \{j' : j' \in S' \setminus T\}$.

This matrix corresponds to the evaluation points in $[n] \setminus T$. However, we will look at a different matrix that corresponds to the evaluation points $\{j' : j' \in S' \setminus T\}$ and $\{-d+\ell, \ldots, 0\}$. Since this matrix is generated by taking linear combinations of rows in $M_{\bar{T},S'}$, this can only decrease the rank. The matrix we obtain has $(d+1-t)+(d-\ell+1) = 2d-\ell-t+2$ rows and $(d-\ell+1)(d+1-t)$ columns, and it looks like this:

$$\begin{pmatrix} 1 & & & 1 & & & \cdots\cdots\cdots & 1 & & \\ & \ddots & & & \ddots & & & & \ddots & \\ & & 1 & & & 1 & \cdots\cdots\cdots & & & 1 \\ * & \cdots & * & & & & & & & \\ & & & * & \cdots & * & & & & \\ & & & & & & \ddots & & & \\ & & & & & & & * & \cdots & * \end{pmatrix},$$

where the $d-\ell+1$ first rows correspond to the evaluation points $\{-d+\ell, \ldots, 0\}$ and the later $d+1-t$ correspond to the points j' in $S' \setminus T$ (and $*$ are non-zero elements). Now, by subtracting the first column from each of the columns with '1' in the first row, and eliminating all the columns with a non-zero coefficient in top row except for the first $d-\ell+1$ columns, we get the following matrix:

$$\begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ * & \cdots & * & -* & \ldots & -* & \\ & & & * & & & \\ & & & & \ddots & & \\ & & & & & * & \end{pmatrix}$$

This matrix has rank $2d-\ell-t+1$ as required.

Finally, observe that we have here $k'(d+1)$ distinct subsets. Each subset is of cardinality $n-d$, and so the number of seeds that are distributed is $k'(d+1)(n-d)$. Finally, from each seed one pseudorandom element is derived for generating $P_1(x)$ and $d-\ell+1$ for generating $R(x)$. Overall, the number calls to a PRF is thus $k'(d+1)((n-d)+(d-\ell+1))$ as required. This concludes the proof. ■