Fast Fully Secure Multi-Party Computation over Any Ring with Two-Thirds Honest Majority

Anders Dalskov
Partisia
Aarhus, Denmark
anderspkd@fastmail.com

Daniel Escudero
J.P. Morgan AI Research
New York, USA
daniel.escudero@protonmail.com

Ariel Nof
Technion
Haifa, Israel
ariel.nof@cs.technion.ac.il

ABSTRACT

We introduce a new MPC protocol to securely compute any functionality over an arbitrary black-box finite ring (which may not be commutative), tolerating $t < n/3$ active corruptions while *guaranteeing output delivery* (G.O.D.). Our protocol is based on replicated secret-sharing, whose share size is known to grow exponentially with the number of parties n. However, even though the internal storage and computation in our protocol remains exponential, the communication complexity of our protocol is *constant*, except for a light constant-round check that is performed at the end before revealing the output.

Furthermore, the amortized communication complexity of our protocol is not only constant, but very small: only $1 + \frac{t-1}{n} < 1\frac{1}{3}$ ring elements per party, per multiplication gate over two rounds of interaction. This improves over the state-of-the art protocol in the same setting by Furukawa and Lindell (CCS 2019), which has a communication complexity of $2\frac{2}{3}$ *field* elements per party, per multiplication gate and while achieving fairness only. As an alternative, we also describe a variant of our protocol which has only one round of interaction per multiplication gate on average, and amortized communication cost of $\leq 1\frac{1}{2}$ ring elements per party on average for any natural circuit.

Motivated by the fact that efficiency of distributed protocols are much more penalized by high communication complexity than local computation/storage, we perform a detailed analysis together with experiments in order to explore how large the number of parties can be, before the storage and computation overhead becomes prohibitive. Our results show that our techniques are viable even for a moderate number of parties (e.g., $n > 10$).

CCS CONCEPTS

• **Security and privacy → Cryptography**.

KEYWORDS

Multiparty Computation; Honest Majority; Robust Computation

ACM Reference Format:
Anders Dalskov, Daniel Escudero, and Ariel Nof. 2022. Fast Fully Secure Multi-Party Computation over Any Ring with Two-Thirds Honest Majority. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559389

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559389

1 INTRODUCTION

Secure Multiparty computation (MPC) is a set of techniques that enables a group of mutually distrustful parties to securely compute a given function on private data, while revealing only the output of the function. MPC protocols provide a general-purpose tool for computing on sensitive data while eliminating single points of failure.

Due to the strong guarantees that MPC protocols provide, together with the wide range of applications that they enable, several real-world problems where computation on sensitive data is required have been solved using MPC techniques. Practical applications have been found so far in key management, financial oversight [1], MPC secured database [11], market design [12], biomedical computations [9, 19] and even satellite collision detection [39].

Since feasibility results for MPC were established in the 80s [8, 18, 35, 46], the problem of constructing efficient protocols for secure computation has gained significant interest. Applications we see today are enabled thanks to a long line of works that have aimed at improving the efficiency of MPC protocols, as well as extending the theory of the field.

It is well known that the efficiency, or even the feasibility of certain MPC protocols, depends heavily on the type of security that is desired. This can be described in different and orthogonal categories:

Who can be corrupted? We assume that the adversary corrupts at most t out of the n participants.[1] Ideally, no requirement besides $t < n$ would be imposed. However, it turns out that, if we require the stronger condition of $t < n/2$, or even $t < n/3$, more efficient protocols with better security guarantees can be devised.

How do the corrupt parties behave? If the corrupt parties follow the protocol specification then the corruption is said to be *passive* (or semi-honest). In contrast, under an *active* (or malicious) corruption, the parties are not required to adhere to the protocol, and they may deviate arbitrarily.

What is the adversary's computational power? Perfect security, which ensures that even a computationally unbounded adversary cannot learn anything about the honest parties' inputs, is only achievable if $t < n/3$. Statistical security allows a negligible probability of leakage, and is only achievable if

[1] The so-called *generalized adversarial structures* allow for a more fine-grained description of the subsets of parties, but in this work we restrict to *threshold adversarial structures*, which are defined by a threshold t.

$t < n/2$. Finally, if $t < n$ then the adversary has to be computationally bounded, as it must not be able to break certain hard computational problems.

Is the adversary allowed to cause an abort? Finally, we can choose if the computation must be guaranteed to terminate, which is called *full security*, or if the adversary can cause an *abort*, perhaps learning the result of the computation before the honest parties do or causing a denial-of-service. The former is only possible if $t < n/2$ given secure point-to-point channels and a broadcast primitive [45] (where the latter can be realized from a public-key infrastructure using digital signatures [31], or alternatively using only secure channels assuming $t < n/3$ [8, 18]).

Different combinations of the categories are either impossible, or lead to protocols with different level of efficiencies. For example, it is well known that working in the dishonest majority setting, where $t < n$, adds a considerable overhead, and requires use of relatively expensive public-key cryptography primitives such as oblivious transfer. Furthermore, full security, or even fairness, is generally not achievable in this setting. In contrast, protocols for honest majority ($t < n/2$) or two-thirds honest majority ($t < n/3$) do not need to make use of computationally expensive cryptography, and they can achieve the strongest notion of full security; this, of course, at the expense of tolerating a weaker adversary corrupting less parties.

A good set of experimental results for different combinations of the categories above can be found in [27]. The main take-away lesson is that the efficiency of MPC protocols, plus the type of guarantees they can provide, depends heavily on different factors. For a practical deployment of MPC, it is necessary to look at the most realistic combination of the "parameters" above, in such a way that the resulting protocol is reasonably efficient and still secure for the application at hand. For example, although it may be reasonable to assume a passive adversary in a restricted set of settings, it is natural to desire security against active adversaries, given that there is no way to audit that a given participant of an MPC protocol is sending messages correctly, so this constitutes an easy way to cheat if there is enough motivation and profit from doing so.

On the other hand, the bound t on the number of corrupted parties is a less clear parameter to set. As mentioned before, the setting $t < n$ is ideal since it guarantees security for every single individual as long as it behaves honestly, but this might be too strong in some scenarios, especially when there is enough diversity among the participants and there is no strong reason to expect a large adversarial coalition to be formed. This, coupled with the inefficiency that protocols in this setting typically suffer from, leads us to naturally consider more lenient adversarial thresholds such as $t < n/2$ or $t < n/3$ that enable much more efficient protocols. Furthermore, many applications benefit from—or outright require—full security, which is only achievable in these threshold regimes. For example applications related to Machine Learning, where the computation is very large and thus very costly to run; or applications related to voting where rerunning the computation is simply not possible. The observations above set the motivation for our work.

1.1 Our Contribution

In this work, we consider the setting of $t < n/3$ with active and full security. Although this adversarial threshold is lower than the weak honest-majority and dishonest-majority settings, extremely efficient protocols can be designed in this regime, having very simple design and a thin layer that adds full security. In this direction, we present a simple new MPC protocol to compute any arithmetic circuit with the following characteristics:

- Full security against an active adversary corrupting at most $t < n/3$;[2]
- Computation over *any* finite ring (even non-commutative ones);
- Communication complexity of $n + t - 1$ ring elements in total per multiplication gate, plus a term which is independent of the number of multiplication gates. This is, the amortized communication cost is $\leq 1\frac{1}{3}$ ring elements per multiplication gate per party.
- Statistical security, except for the use of a PRG to boost efficiency.

Our protocol works by computing the circuit using a passive protocol which guarantees only privacy, and then verifying the correctness of the computation using a novel sub-protocol which incurs only constant communication cost (independent of the size of the computed circuit) and constant number of rounds. The only drawback of our protocol is that, due to the use of replicated secret sharing as the underlying secret sharing scheme, local storage and computation grows exponentially with the number of parties. However, since this does not affect communication of the underlying passive protocol, it is only for larger values of n that this weakness starts to kick in. In Section 6 we assess experimentally the feasibility of our protocol for a reasonably large number of parties. We show that replicated secret sharing-based protocols are not restricted in practice to only a very small number of parties (such as 3 or 4), as traditionally believed. We remark that if one is willing to settle for security with abort only, then this restriction can be removed, as we use the properties of replicated secret sharing only to identify cheaters. When considering security-with-abort only our protocol can therefore work with Shamir's secret sharing and achieve the same efficiency as the state-of-the-art protocol in this setting of Furukawa and Lindell [33]. However, unlike their work, we are able to augment our protocol to full security which is much stronger.

Our basic protocol requires 2 rounds of interaction per multiplication gate. As an alternative, we present a variant of our protocol with only a *single* round per multiplication, at the expense of increasing the communication cost slightly. In particular, for "natural" circuits, where the gate can be divided into groups, where output wires from gates in one group only enters gates from the second group, the communication cost is ≤ 1.5 sent ring elements per multiplication gate for each party. This variant is described in the full version of the paper.

Moreover, for $n = 4$ and $t = 1$, which is the base case of this setting, our protocol incurs communication of 1 ring element per party, distributed across two rounds. When using the variant with only one round of interaction, the cost increases to just $\frac{9}{8}$ elements (for natural circuits as defined above). This improves upon the recent protocols that were designed only for this setting [28, 36, 41],

[2] Our protocol can be easily generalized to arbitrary Q3 adversarial structures.

which requires communication of 1.5 ring elements and a single online round of interaction, or the more recent work of [42], which uses $1\frac{1}{4}$ elements per party, also in one online round. Furthermore, our protocol enjoys a very simple design that generalizes to any number of parties beyond $n = 4$.

Finally, our protocol works over any ring (even non-commutative ones) in a *black-box* way. This is in stark contrast with essentially all prior work (we elaborate on this point in Section 1.2), which rely on rings that are commutative, or have "high invertibility", like finite fields. As a result, our protocol can operate *natively* over relevant non-commutative rings such as matrix rings, which are widely used in settings like machine learning (e.g. neural networks, support vector machines, linear regression, etc.).[3] In addition, commutative rings such as integers modulo 2^k, which have received quite some attention recently [2, 3, 20, 24, 29, 44], are also encompassed by our protocol.

1.2 Related Work

The goal of achieving linear communication complexity (in the number of parties) and with prefect security when $t < n/3$ was obtained in [6, 37]. The protocol of [6] was used in a more practical setting in [5] by settling for security with abort only. Later, this was improved by [33], leading to a computationally-secure protocol with fairness in which each party sends, on average, $2\frac{2}{3}$ field elements. As explained above, we improve over [33] by achieving full security.

Several works have focused on achieving full security in the setting of $n = 4$ and $t = 1$ [28, 36, 41]. The state-of-the-art protocol by Koti *et al.* [42] requires sending 1.25 ring elements per multiplication per party in one online round, which we improve for any natural circuit. It should be noted that [42] also provide a protocol for 3-input multiplication gates with 3 ring element sent per party in one online round, which we did not consider in this work.

In the setting of $t < n/2$ recent breakthrough results have shown how to achieve full security with low communication. The protocol of [38] requires each party to send 5.5 field elements per multiplication with information-theoretic security, while the protocol of [16] reduces communication to 1.5 ring elements by allowing use of any PRG. For $n = 3$ and $t = 1$, communication can be further reduced to 1 ring elements as shown in [15]. While the later protocols achieve similar amortized communication as ours with a more powerful adversary, our protocol has several advantages over theirs. First, the additive overhead to achieve active security in these protocols is *logarithmic* in the size of the circuit while ours is *constant*. The same applies to the number of calls to an expensive broadcast channel which is logarithmic in the circuit's size in these protocols and constant in ours. Furthermore,for the case of the ring $\mathbb{Z}_{2^k}$, their protocol requires arithmetic over Galois ring extensions of very large degree (> 50), whose concrete efficiency is unclear (see for example [28]). The above is due to the fact that they rely on distributed zero-knowledge proofs [13], which we are able to avoid in our setting. Finally, achieving robustness in the setting of $t < n/2$ requires using authentication tags which makes these protocols much more complicated and computationally expensive compared to our protocol which avoids this completely. Without full security, for example with security with abort, efficient protocols exist in the honest majority setting (e.g. [21] and [3]).

We also point out that replicated secret-sharing for an arbitrary number of parties has been already used in works like [4]. However, in their protocol, communication per gate grows exponentially with the number of parties, whereas in our protocol the communication cost per gate is constant.

Finally, the feasibility of computation over general rings was shown in [26]. The protocol from [6] was generalized to the ring of integers modulo 2^k in [2]. Furthermore, the 4-party protocols of [28, 36, 41] also work over this ring. In contrast, the work of [33] focus on multiparty computation over finite fields. We are thus not aware of a concretely-efficient work[4] in this setting for more than $n = 4$ that applies to general rings. More recently, [32] considers MPC for arbitrary circuits over black-box finite rings, which could be potentially non-commutative. However, their results are mostly of theoretical interest since, due to the lack of commutativity, the offline phase in their protocols results in a large overhead. Nevertheless, we remark that their local computation, unlike ours, is polynomial in the number of parties.

2 PRELIMINARIES

Notation. Let κ be the security parameter and let n be the number of parties. We denote the set of involved parties by $\mathcal{P} = \{P_1, \ldots, P_n\}$. Let t be an upper bound in the number of corrupted parties, and assume that $t < n/3$. Many of our subprotocols will be presented with a threshold $d \leq t$, since, due to the player elimination framework, they may be executed with a smaller threshold than t. We use the notation $[n]$ for the set $\{1, \ldots, n\}$.

2.1 Background in Ring Theory

Let R be *any* finite ring. We only assume procedures for adding and multiplying ring elements, as well as sampling uniformly random elements. A set $A \subseteq R$ is called *exceptional* if, for all $x, y \in A$ with $x \neq y$, $x - y$ is invertible.[5]

For the rest of the paper, let $\mathbb{A}$ be the any of the largest exceptional subsets of R, and let $\omega_R = |\mathbb{A}|$. We will need the following lemma in our protocol (the proof can be found in the full version):

Lemma 1. *Let $a, b \in R$, with $a \neq 0$. Then:* $\Pr_{x \xleftarrow{\$} \mathbb{A}}[x \cdot a + b = 0] \leq \frac{1}{\omega_R}$.

Observe that if R is a field then we may take $\mathbb{A} = R$, and therefore $\omega_R = |R|$. On the other hand, if R is the ring of integers modulo 2^k, it can be shown that there are no exceptional sets of size 3 or more, so we may take $\mathbb{A} = \{0, 1\}$, and hence $\omega_R = 2$.

2.2 MPC Security Definition

In this work, we consider adversaries who can follow an arbitrary strategy to carry-out their attack. We use the standard ideal/real paradigm [34] in order to define security, where an execution in the

[3] Furthermore, for the particular case of matrix rings, our secret-sharing scheme enables local conversions between shared matrices and "entry-wise" sharings, which is essential for many applications like the ones described above, as they typically manipulate individual entries along with the matrix arithmetic. This is not possible for example with the work of [32].

[4] By "concretely-efficient", we mean protocols with low communication that use only cheap symmetric crypto.

[5] In a finite non-commutative ring, a is invertible if there exists b such that $a \cdot b = b \cdot a = 1$.

ideal world with a trusted party who computes the functionality for the parties is compared a real execution. Although our protocols can be computed with information-theoretic security, we use minimal computational assumptions to achieve better concrete efficiency. Thus, when we say that a protocol "computationally computes" an ideal functionality, this means that the output of the ideal execution with an ideal world simulator is *computationally indistinguishable* from the output of the real world execution. In some of our protocols, there is also a statistical error, which is independent of the computational security parameter. As in [33], we formalize security in this case by saying that the outputs of the two executions can be distinguished with probability of at most some negligible function in the security parameter, *plus* the statistical error.

We prove the security of our protocols in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. The modular sequential composition theorem of [17] states that one can replace the trusted party computing the subfunctionality with a real secure protocol computing the subfunctionality. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

Universal Composability. Protocols that are proven secure in the universal composability framework have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In [43], it was shown that any protocol that is proven secure with a black-box non-rewinding simulator and also has the property that the inputs of all parties are fixed before the execution begins (called input availability or start synchronization in [43]), is also secure under universal composability. Since the input availability property holds for all of our protocols and subprotocols, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability from [43]. This also enables us to call the protocol and subprotocols that we use in parallel and concurrently (and not just sequentially), allowing us to achieve more efficient computation (i.e. by running many executions in parallel or running each layer of a circuit in parallel).

2.2.1 Broadcast and Agreement. A protocol for Byzantine agreement takes a bit from all parties as an input, and let all honest parties reach a consensus in the presence of t corrupted parties. If all honest parties holds the same bit b, then the protocol guarantees that all honest parties will output b. In our protcol, we will use Byzantine agreement to let a party broadcast one bit to the other parties. This can be done by letting the party send the bit to all parties and then run Byzantine agreement. In our setting of $t < n/3$, perfect Byzantine agreement can be achieved with quadratic communication complexity [10, 23]. We stress that the number of calls to broadcast in our protocol is *constant* and so any way to implement it suffices.

3 REPLICATED SECRET SHARING AND ITS OPERATIONS

Replicated secret sharing. The replicated secret sharing scheme [40], with threshold $d \leq t$, is defined by the following procedures. Below, we let $\lambda = \binom{n}{d}$ and let $T_1, \ldots, T_\lambda \subset \mathcal{P}$ be all subsets of parties of size $n - d$.

- share(x, d): To share a secret x with threshold d, the dealer generates λ random $x_{T_1}, \ldots, x_{T_\lambda} \in R$ under the constraint that $x = x_{T_1} + \cdots + x_{T_\lambda}$. Then, the dealer hands x_{T_j} to the parties in T_j. The share $\vec{x}_i$ held by party P_i is a tuple consisting of all x_{T_j} such that $P_i \in T_j$. We say that $[\![x]\!]_d$ is the collection of all $\vec{x}_i$s.
- reconstruct$([\![x]\!]_d, i)$: In this interactive procedure, the parties in each subset T where $|T| = n-d$ and $P_i \notin T$, send all their shares to P_i. For each subset of parties T holding a share x_T, if P_i received different values for x_T, then P_i takes the *majority value* to be x_T. Finally, P_i sets $x = \sum_{T \subseteq \mathcal{P}:\ |T|=n-d} x_T$.

Secrecy of this scheme follows from the fact any set of d corrupted parties miss one additive share (namely, the one indexed by their complement), and so the secret could be any value in the ring.

Note that the sharing procedure described above implies that a corrupted dealer may cheat by sending different values to different parties in the same subset T of parties. In this case, we say that the sharing is inconsistent. We formally define the notion of consistency in the following definition.

Definition 2 (Consistency). We say that $[\![x]\!]_d$ is consistent if for each two honest parties P_i and P_k, for each $T \subset \mathcal{P}$, such that $|T| = n - d$ and $P_i, P_k \in T$, it holds that the same x_T is held by both P_i and P_k.

Relying on the definition of consistency, we next prove that the reconstruct procedure defined above is robust, i.e., the receiving party will always obtain the correct secret.

Claim 3. *If $[\![x]\!]_d$ is consistent and $d < \frac{n}{3}$, then* reconstruct$([\![x]\!]_d, i)$ *ends with P_i holding x, even in the presence of malicious adversaries controlling up to d parties.*

Proof: In the procedure, P_i receives from all parties in each subset T of $n - d$ parties with $P_i \notin T$, the share x_T. Since $3d < n$, we have that $n - d \geq 2d + 1$. This implies that in each subset, there is a majority of honest parties. Since $[\![x]\!]_d$ is consistent, it means that all honest parties in each T will send the same value, and so by taking the majority value, P_i will obtain the correct share. ∎

Complexity. For n parties and threshold d, there are $\binom{n}{d}$ distributed shares. Each party holds $\binom{n-1}{d}$ shares. When reconstructing a secret towards P_i, party P_i receives $\binom{n-1}{d-1}$ missing shares, and each share is sent by $n - d$ parties. Thus, the overall communication is $\binom{n-1}{d-1} \cdot (n - d) = \binom{n-1}{d} \cdot d$ ring elements.

Pairwise consistency check. The above definition gives us an easy way to check that a sharing is consistent, by having each pair of parties comparing their joint shares. Note that each pair of parties can check pairwise consistency of an arbitrarily large number of sharings by comparing a hash of the string consisting of all their joint shares.

3.1 Local Operations

Linear operations. Let $[\![x]\!]_d$ and $[\![y]\!]_d$ be two consistent sharings and let $\alpha \in R$ be some public constant. We define the following operations:

- $[\![x]\!]_d + [\![y]\!]_d$: let $\vec{x}_i$ and $\vec{y}_i$ be the two vectors of shares held by P_i. Then, P_i performs point-wise addition between the two vectors and stores the result as its output.
- $\alpha \cdot [\![x]\!]_d$: let $\vec{x}_i$ be the vector of shares held by P_i. Then, P_i multiplies each component in $\vec{x}_i$ with α and store the result as its output.
- $\alpha + [\![x]\!]_d$: One pre-determined subset of parties T with $|T| = n-d$ which holds x_T define $x_T \leftarrow x_T + \alpha$. The other $\lambda - 1$ shares (where $\lambda = \binom{n}{d}$) remains the same.

The next claim is straight-forward given the definitions of the operations:

Claim 4. *For every $[\![x]\!]_d$ and $[\![y]\!]_d$ and a constant $\alpha \in R$ it holds: (i) $[\![x+y]\!]_d = [\![x]\!]_d + [\![y]\!]_d$; (ii) $[\![\alpha \cdot x]\!]_d = \alpha \cdot [\![x]\!]_d$; (iii) $[\![\alpha + x]\!]_d = \alpha + [\![x]\!]_d$.*

Multiplication. We next show two local operations for multiplying two shared inputs x and y, in order to generate a sharing of $x \cdot y$, but with *a higher threshold*. The first operation, which we denote by $[\![x]\!]_d \cdot [\![y]\!]_d$, aims to generate $[\![x \cdot y]\!]_{2d}$. The second operation, which we denote by $[\![x]\!]_d \odot_U [\![y]\!]_d$, aims to compute an *additive* sharing of $x \cdot y$ across a pre-determined subset of parties $U \subseteq \mathcal{P}$ of size $n - d$. We denote such a sharing by $\langle x \cdot y \rangle^U$.

Recall that $x = x_1 + \cdots + x_\lambda$ and $y = y_1 + \cdots + y_\lambda$. It follows that $x \cdot y = \sum_{j \in [\lambda]} x_j \cdot \sum_{k \in [\lambda]} y_k = \sum_{j,k \in [\lambda]} x_j \cdot y_k$. This implies that in order to locally generate a sharing of $x \cdot y$, we need each product $x_j \cdot y_k$ to be known by a set of parties of a sufficient size, where the set's size is determined by the desired threshold. In our setting of $d < \frac{n}{3}$, this indeed holds and utilized in the following two procedures:

- $[\![x]\!]_d \cdot [\![y]\!]_d$:
  - For each $T \subset \mathcal{P}$ such that $|T| = n - 2d$: The parties in T initialize $z_T := 0$.
  - For each pair x_j, y_k that are known to a set of parties $S \in \mathcal{P}$: let $T \subset S$ be the subset containing the first $n - 2d$ parties in S and let $q = \binom{|S|}{n-2d}$.
    Then, the parties in T set: $z_T \leftarrow z_T + q \cdot (x_j \cdot y_k)$.
    For each $T' \subset S$ with $|T'| = n - 2d$ and $T' \neq T$ set: $z_{T'} \leftarrow z_{T'} - (x_j \cdot y_k)$.
  - Each party P_i sets $\vec{z}_i$ to be the tuple of all z_T for which $P_i \in T$ with $|T| = n - 2d$, and stores it as its output.
- $[\![x]\!]_d \odot_U [\![y]\!]_d$:
  - Each party $P_i \in U$ initializes $z_i := 0$.
  - For each pair of x_j and y_k, let $T \subset U$ be the set of parties that holds both x_j and y_k and let P_ℓ be the party with the smallest index in T. Then, P_ℓ sets: $z_\ell \leftarrow z_\ell + |T| \cdot (x_j \cdot y_k)$, whereas each $P_u \in T$ with $u \neq \ell$ sets: $z_u \leftarrow z_u - (x_j \cdot y_k)$.
  - Each party $P_i \in U$ stores z_i as its output.

Claim 5. *Let $d \in \mathbb{N}$ be such that $n > 3d$ and let $U \subseteq \mathcal{P}$. If $|U| \geq 2d + 1$, then for every two sharings $[\![x]\!]_d$ and $[\![y]\!]_d$ it holds: (i) $[\![x \cdot y]\!]_{2d} = [\![x]\!]_d \cdot [\![y]\!]_d$; (ii) $\langle x \cdot y \rangle^U = [\![x]\!]_d \odot_U [\![y]\!]_d$.*

The proof can be found in the full version.

3.2 Non-Interactive Random Secret Generation

We next show how to generate correlated randomness required by our protocol, without any interaction, but a short set-up step. Let $\mathcal{F} = \{F_k \mid k \in \{0,1\}^\kappa, F_k : \{0,1\}^\kappa \to R\}$ be a family of pseudo-random functions. The parties work as follows:

From $[\![k]\!]_d$ to any number of $[\![r]\!]_d$. As shown in [25], given $[\![k]\!]_d$, the parties generate the ℓth random sharing $[\![r_\ell]\!]_d$, by having each subset T holding k_T, set its share to be $r_\ell^T = F_{k_T}(\ell)$.

The ideal functionality $\mathcal{F}_{\text{coin}}$. Let $\mathcal{F}_{\text{coin}}$ be an ideal functionality that hands the parties fresh random coins. It can be securely realized by having the parties compute a new random sharing $[\![r]\!]_d$ and then open it by running reconstruct$([\![r]\!]_d, i)$ for each $i \in [n]$. To generate any numbers of random coins with a constant cost, the parties can use the above procedure to generate a new key k from which all the required randomness is derived using a pseudo-random function F_k.

3.2.1 $\mathcal{F}_{\text{zeroShare}}$ - Generating any number of $[\![0]\!]_{2d}$. In our protocol, we will need random sharings of 0 with threshold $2d$, which will be used to randomize given sharings. We show how the parties can generate any number of $[\![0]\!]_{2d}$ from a single sharing $[\![k]\!]_d$ without any interaction. To generate the ℓth $[\![0]\!]_{2d}$ sharing, the parties work in the following way:

(1) Each subset $S \in \mathcal{P}$ of size $n - 2d$ initializes $r_S = 0$.
(2) For every subset $T \in \mathcal{P}$ of size $n - d$, holding k_T:
    Let $\theta = \binom{n-d}{n-2d}$ and let $S^1, S^2, \ldots, S^\theta$ be all subsets of size $n - 2d$ in T. Then:
    For each $j \in \{2, \ldots, \theta\}$, the parties in S^j set: $r_{S^j} \leftarrow r_{S^j} - F_{k_T}(\ell \| j)$.
    The parties in subset S^1 set: $r_{S^1} \leftarrow r_{S^1} + \sum_{j=2}^{\theta} F_{k_T}(\ell \| j)$.
(3) Each party P_i outputs a vector all r_S for which $P_i \in S$.

Note that each $F_{k_T}(\ell \| j)$ is added once and subtracted once, and so overall $\sum_{S \in \mathcal{P}: |S| = n-2d} r_S = 0$ as required.

To prove security, let $\mathcal{F}_{\text{zeroShare}}$ be an ideal functionality that receives from the adversary a share for each subset of parties of size $n - 2d$ that contains corrupted parties, and then chooses random shares for the remaining subsets (which contain honest parties only), under the constraint that all shares will sum to 0. Then, $\mathcal{F}_{\text{zeroShare}}$ sends the honest parties their shares. We thus have the following (the proof can be found in the full version of this paper):

Lemma 6. *If F_k is a pseudo-random function, then our protocol as described in the text, computationally computes $\mathcal{F}_{\text{zeroShare}}$ in the presence of any malicious adversary controlling up to d parties, where $d < \frac{n}{3}$.*

3.2.2 $\mathcal{F}_{\text{corRand}}$ - Generating any number of $([\![r]\!]_d, \langle r \rangle^U)$ for a pseudo-random $r \in R$ and $U \subset \mathcal{P}$ such that $|U| = 2d + 1$. Recall that $\langle r \rangle^U$ is an additive sharing of r across the parties in U. To obtain pairs of sharings of the same r, the traditional approach is to generate two sharings with the two thresholds separately, and then check that the obtained sharings are of the same secret. We use a different approach, where each party in U first chooses its additive share, shares it to all the other parties, and then the parties use these to compute the sharing with threshold d. Besides the fact that it allows us to avoid the need to run a check, we obtain here a property that will be used later: *the additive share of each party in U is robustly shared to the other parties.* Formally, the parties work as follow:

- *Setup step:*
  (1) Each party $P_i \in U$ chooses a random $k_i \in R$ and shares $[\![k_i]\!]_d$ to the parties.
  (2) The parties run pair-wise consistency check. If party P_j finds that the shares held by him and P_k are not the same, then it broadcasts (inconsistent, j, k). Then, P_i broadcasts all shares that are held by subsets that contain both P_j and P_k.
  (Note that since in this case either P_i, P_k or P_j is corrupted, these shares are anyway known to the adversary, and so publishing them gives the adversary no additional information.)
- *Generating the ℓth pair* $([\![r_\ell]\!]_d, \langle r_\ell \rangle^U)$*:*
  (1) For each i with $P_i \in U$: the parties compute $[\![r_{\ell,i}]\!]_d$ from $[\![k_i]\!]_d$ as shown above. Knowing all shares, party P_i computes $r_{\ell,i}$ and sets it as its additive share of r_ℓ.
  (2) The parties locally compute $[\![r_\ell]\!]_d = \sum_{P_i \in U} [\![r_{\ell,i}]\!]_d$.

Observe that $r_\ell = \sum_{P_i \in U} r_i$ and so the parties hold an additive sharing and a d-out-of-n sharing of r_ℓ as required. In addition, as promised above, the additive share r_i of each party $P_i \in U$ is shared to the other parties via a d-out-of-n secret sharing. This property will be used later in our protocol.

The $\mathcal{F}_{\text{corRand}}$ ideal functionality. In our protocol, each time the parties will need correlated randomness from the type defined above, they will call the $\mathcal{F}_{\text{corRand}}$ ideal functionality defined in Functionality 7. The functionality $\mathcal{F}_{\text{corRand}}$ lets the adversary choose the shares of the corrupted parties, and then chooses random share for honest parties, under the constraint that the same secret r is stored in $\langle r \rangle^U$ and $[\![r]\!]_d$.

FUNCTIONALITY 7 (The $\mathcal{F}_{\text{corRand}}$ ideal functionality).
The $\mathcal{F}_{\text{corRand}}$ ideal functionality works with an ideal world adversary $\mathcal{S}$ and honest parties. Let $I \subset [n]$ with $|I| \leq d$, be the set of the corrupted parties' indices, and $H = [n] \setminus I$ be the set of the honest parties' indices. Finally, let $U \in P$ be a predetermined set of parties.

(1) $\mathcal{F}_{\text{corRand}}$ receives $\{r_i\}_{i \in I: P_i \in U}$ and $\{r_T\}_{T \subset \mathcal{P}:\ (|T|=n-d) \wedge (\exists i \in I: P_i \in T)}$ from $\mathcal{S}$.
(2) $\mathcal{F}_{\text{corRand}}$ chooses a random $r_j \in R$ for each $j \in H$ such that $P_j \in U$, and sets $r = \sum_{k=1}^{n} r_k$. Then, it chooses a random $r_T \in R$ for each $T \subset \mathcal{P}$ with $|T| = n - d$ that contains only honest parties, under the constraint that $r = \sum_{T \subset \mathcal{P}:\ |T|=n-d} r_T$.
(3) For each honest party P_j, $\mathcal{F}_{\text{corRand}}$ hands $\{r_T\}_{P_j \in T}$ to P_j and, if $P_j \in U$, hands also r_j to P_j.

Lemma 8. *If F_k is a pseudo-random function, then our protocol as described in the text, computationally computes $\mathcal{F}_{\text{corRand}}$ in the presence of any malicious adversary controlling up to d parties, where $d < \frac{n}{3}$.*

The proof can be found in the full version.

4 BUILDING BLOCKS

In this section, we outline three sub-protocols that are used in our main protocol: a protocol for multiplying shared inputs which achieves only privacy, a protocol to verify the correctness of many multiplication triples and a protocol for eliminating corrupted parties from the computation.

4.1 Multiplying Two Shared Values - The DN Protocol [30]

The Damgård-Nielsen protocol [30] is the fastest multiplication protocol in the honest majority setting, known to this date. The "text-book" version of this protocol, requires the parties to prepare in advance a pair of random sharings $[\![r]\!]_d, [\![r]\!]_{2d}$. Then, in order to multiply $[\![x]\!]_d$ and $[\![x]\!]_d$, the parties locally compute $[\![x]\!]_d \cdot [\![y]\!]_d - [\![r]\!]_{2d}$ and send their shares to P_1 (or any other designated party). Party P_1 reconstructs $xy - r$ and sends it back to the parties. Then, the parties locally compute $[\![x \cdot y]\!]_d = [\![r]\!]_d + (x \cdot y - r)$.

While the protocol was constructed and used throughout the years for Shamir's secret sharing scheme, a simple observation made by [14] is that all operations required by this protocol can be carried-out also when using replicated secret sharing.

Achieving privacy in the presence of malicious adversaries. The text-book version of the DN protocol described above is semi-honest secure. A somewhat surprising finding by [37] is that it actually *does not* achieve even privacy in the presence of malicious adversaries when $d < n/3$. In particular, a malicious P_1 can learn intermediate values. The attack is carried-out over two gates in two preceding layers. Assume that the parties need to multiply $[\![x]\!]_d$ and $[\![y]\!]_d$. Then, upon receiving the messages from the parties and computing $x \cdot y - r$, party P_1 sends the correct value to all parties, except for one honest party, say P_n, to whom it sends $x \cdot y - r + 1$. This implies that P_n now holds an incorrect share of the output $x \cdot y$. Next, assume that in the next layer $x \cdot y$ is being multiplied with w. The crux of the attack is that any $n - 2d$ shares of $xy \cdot w - r'$ (where r' is the random secret mask used in this gate) determine deterministically the remaining shares. Thus, upon holding $n - 2d$ shares of $xy \cdot w - r'$, which do not include shares held by P_n, party P_1 can compute the correct shares of P_n. However, P_n will send shares of $(xy + \epsilon) \cdot w - r'$. Thus, by taking the difference between the actual share received from P_n and the share that should have been sent, P_1 can learn private information about P_n's shares of w.

As can be seen from the above description, the main reason behind the attack is the fact that the random masking sharing $[\![r]\!]_{2d}$ has redundancy, allowing P_1 to use $n - 2d$ shares to compute the remaining $2d$ shares. Thus, a simple way to prevent this attack is to use a mask that is additively shared between the parties. To reduce communication, it suffices to use an additive sharing across $2d + 1$ parties (including P_1) only. This means that the parties need to prepare a pair of sharings $[\![r]\!]_d, \langle r \rangle^U$ for each multiplication, where U is a set of $2d + 1$ parties which includes P_1, and locally compute a sharing $\langle x \cdot y \rangle^U$ which is opened towards P_1. Fortunately, as shown in Section 3.2, we are able to generate this type of correlated randomness. In addition, replicated secret sharing allows computing $\langle x \cdot y - r \rangle^U$ given $[\![x]\!]_d, [\![y]\!]_d$ and $\langle r \rangle^U$, by taking $[\![x]\!]_d \odot_U [\![y]\!]_d$ (see the definition of this operation in Section 3.1) which yields $\langle x \cdot y \rangle^U$, and then subtracting $\langle r \rangle^U$.

Reducing communication. A simple optimization to the DN protocol that we can apply when using replicated secret sharing, is letting P_1 send $x \cdot y - r$ to only *one* subset of parties of size $n - d$. This suffices since adding $x \cdot y - r$ to $[\![r]\!]_d$, is by definition (see Section 3.1) carried-out by having one subset adding $xy - r$ to their share of r. If this subset includes P_1, then it follows that P_1

needs to send $n-d-1$ ring elements in the second round of the protocol. Overall, the number of elements sent in the protocol is $2d+n-d-1 = n+d-1$, and so per party the cost is $1+\frac{d-1}{n}$ sent ring elements. For $d=1$, this yields 1 ring element per party, and in general, for $d < n/3$ this is bounded by $1\frac{1}{3}$ elements per party.

Formal description. We present a formal description of the protocol in Protocol 9.

PROTOCOL 9 (The Optimized DN Multiplication Protocol).
Let U be the set $\{P_1, \ldots, P_{2d+1}\}$.

- **Inputs:** The parties hold $[\![x]\!]_d$, $[\![y]\!]_d$
- **Set up:** The parties call $\mathcal{F}_{\text{corRand}}$ to obtain $([\![r]\!]_d, \langle r\rangle^U)$.
- **The protocol:**
  (1) The parties in U locally compute $\langle x\cdot y - r\rangle^U = [\![x]\!]_d \odot_U [\![y]\!]_d - \langle r\rangle^U$ and send the result to P_1.
  (2) P_1 reconstructs $xy-r$. Let $T \in \mathcal{P}$ be a predetermined subset of size $n-d$ such that $P_1 \in T$. Then, P_1 sends $xy-r$ to the parties in T.
  (3) The parties compute $[\![z]\!]_d = [\![r]\!]_d + xy - r$ and store $[\![z]\!]_d$ as the output.

The fact that this protocol is private in the presence of malicious adversaries controlling up to d parties, follows easily from the fact that each message sent by an honest party is masked by a new independent random additive mask. Formally, this means that the view of the adversary during the execution have the same distribution, regardless of the honest parties' inputs. Let $\mathcal{F}_{\text{priv}}$ be a protocol to compute a circuit C, where each party shares its input to the other parties, and then the parties traverse over the circuit with topological order, computing multiplication gates using Protocol 9. Let $\text{view}^{f}_{\mathcal{A},\mathcal{F}_{\text{priv}},I}(\vec{v})$ be the view of an adversary $\mathcal{A}$ (i.e., its randomness, inputs, incoming and outgoing messages during the execution) controlling a subset I, when computing a functionality f using $\mathcal{F}_{\text{priv}}$, over a vector of inputs I, *without the output revealing step*. We thus have the following:

Proposition 10. *Let f be a n-aty functionality represented by an arithmetic circuit C over a ring R. Then, for every adversary $\mathcal{A}$ controlling a subset of parties $I \in \mathcal{P}$ with $|I| \le d < \frac{n}{3}$, and for every two vector of inputs $\vec{v}_1, \vec{v}_2$ it holds that* $\text{view}^{f}_{\mathcal{A},\mathcal{F}_{\text{priv}},I}(\vec{v}_1) = \text{view}^{f}_{\mathcal{A},\mathcal{F}_{\text{priv}},I}(\vec{v}_2)$

Remark 11. *If the parties hold two vectors of shares $[\![x_1]\!]_d, \ldots, [\![x_m]\!]_d$ and $[\![y_1]\!]_d, \ldots, [\![y_m]\!]_d$ and wish to compute $[\![\sum_{j=1}^m x_j \cdot y_j]\!]_d$, they can do so without calling the multiplication protocol m times. Instead, this can be done at the cost of one single multiplication, as follows. The parties can locally compute $\langle \sum_{j=1}^m x_j\cdot y_j - r\rangle^U$ by taking $\sum_{j=1}^m ([\![x_j]\!] \odot_U [\![y_j]\!]) - \langle r\rangle^U$, and send the result to P_1, who reconstructs $\sum_{j=1}^m x_j\cdot y_j - r$, send it to the parties, which locally compute $[\![r]\!]_d + \sum_{j=1}^m x_j\cdot y_j - r$. The verification protocol below can be easily adapted to incorporate this operation.*

4.2 Verifying Correctness of Multiplications with Cheating Identification

In this section, we show how the parties can verify that all multiplications were carried out correctly. Our protocol has the property that if someone has cheated, then the parties will detect it with high probability and, in this case, output semi-corrupt pair to eliminate. A pair of parties is called "semi-corrupt" if it contains at least one corrupted party.

The idea behind our protocol is that the parties "compress" the transcript of all multiplication protocols into one single transcript and then verify its correctness. Observe that in our multiplication protocol, all messages are public; the only reason for communication through P_1 is to save communication, and in fact each of the messages could have been sent to all parties. Thus, the first step of our protocol is to agree on the transcript. In this step, the parties sample random coefficients and broadcast a linear combination of the messages they sent and received in all multiplications. If there is conflict between the view of two parties, then a semi-corrupt pair has been found. If all views are consistent, then the parties proceed to the next step, where they verify the correctness of each compressed message. In more details:

Step 1: Agree on the transcript. Let m be the number of multiplications in the circuit. The parties first call $\mathcal{F}_{\text{coin}}$ to receive m random elements $\delta_1, \ldots, \delta_m \in \mathbb{A}$ (this can be done with small constant cost by calling $\mathcal{F}_{\text{coin}}$ to receive a random seed from which all randomness is derived). Then, each party broadcasts a random linear combination of the messages it sent and a random linear combination of the messages it received. Note that each P_i with $i \neq 1$, needs to broadcast one sent message (a linear combination of the messages sent to P_1 in the first round) if it is included in U, and, if it is included in the subset T of parties that receive the message in the second round, broadcast one received message (a linear combination of the messages received from P_1). At the same time, P_1 broadcasts a random linear combination of all messages received from each of the other parties and a random linear combination of the messages it sent in the second round. If there is an inconsistency between a "compressed" message P_1 claims to send/receive to/from P_i and the "compressed" message P_i claims to receive/send from/to P_1, then (P_1, P_i) is the new semi-corrupt pair. The fact that either P_1 or P_i is corrupted holds, since a contradiction cannot occur between two honest parties. If all messages are consistent, then the parties proceed to the next step with an agreed-upon compressed transcript.

Step 2: Verify each party's message. Next, the parties verify the correctness of each party's message. Observe that P_1's message is computed by summing all messages received from the other parties and his own first round's message (since P_1 sees an additive sharing of the masked output). Thus, given the message sent from P_1 in the second round, the parties can compute implicitly the message P_1 would send in the first round. This implies that the verification in this step is reduced to checking that the local computation of $[\![x]\!]_d \odot_U [\![y]\!]_d - \langle r\rangle^U$, performed by each of the parties in U, is correct. Recall that in this computation, each party performs a local computation over its shares of x and y, and then subtracts its additive share of r. Looking at the definition of the operation $[\![x]\!]_d \odot_U [\![y]\!]_d$ from Section 3.1, we observe that in this computation, the parties compute a *linear combination* of their shares. Let $\gamma = \binom{n-1}{n-d-1}$ be the number of shares held by each party. Denote the shares held by P_i by $x_1^i, \ldots, x_\gamma^i$ and $y_1^i, \ldots, y_\gamma^i$ and r^i. Then, party P_i

carries-out the computation $\sum_{k=1}^{\gamma}\sum_{j=1}^{\gamma}(\alpha_{k,j}\cdot x_k^i\cdot y_j^i)-r^i$, where $\alpha_{k,j}$ is a public known coefficient (see Section 3.1).

Next, recalling that in our secret sharing scheme, each share is held by $n-d$ parties, we define $[\![x_k^i]\!]_d$ (and likewise $[\![y_j^i]\!]_d$) to be a consistent d-out-of-n secret sharing of x_k^i in the following way: The subset T of parties which know x_k^i set $x_{k,T}^i = x_k^i$, whereas the other subsets define their share to be 0. In addition, recall that in Section 3.2, we generated the correlated randomness in a special way, such that each party's additive share r^i is also secret shared across the parties in a robust way. We thus have the following:

Fact 12. *In our multiplication protocol (Protocol 9), the message each party sends in the first round, is a result of a degree-2 computation over inputs that are shared across the parties via a consistent d-out-of-n secret sharing.*

Relying on Fact 12, we can thus ask the parties to jointly compute a $2d$-out-of-n secret sharing of P_i's additive share of $x\cdot y-r$, and then compute a $2d$-out-of-n secret sharing of the "compressed message" obtained in the previous step. That is, for each $i\in U$, the parties will compute

$$[\![\mathsf{msg}^i]\!]_{2d} = \sum_{\ell=1}^{m}\delta_\ell\cdot[\![(x_\ell\cdot y_\ell - r_\ell)^i]\!]_{2d} \tag{1}$$

$$= \sum_{\ell=1}^{m}\delta_\ell\cdot\left(\sum_{k=1}^{\gamma}\sum_{j=1}^{\gamma}(\alpha_{k,j}\cdot[\![x_{\ell,k}^i]\!]_d\cdot[\![y_{\ell,j}^i]\!]_d) - [\![r_\ell^i]\!]_d\right) + [\![0]\!]_{2d}$$

where $[\![0]\!]_{2d}$ is a secret sharing of 0 which is added to randomize the parties' shares (and is handed to the parties by $\mathcal{F}_{\text{zeroShare}}$). Note that this computation is completely non-interactive: the parties can locally compute $[\![x_k^i\cdot y_j^i]\!]_{2d} = [\![x_k^i]\!]_d\cdot[\![y_j^i]\!]_d$ and $[\![r^i]\!]_{2d} = [\![r^i]\!]_d\cdot[\![1]\!]_d$ (where $[\![1]\!]_d$ is some known sharing of 1) as defined in Section 3.1, and then locally perform addition and multiplication with the public constants. It remains to open the shared secret and check whether it equals to the additive share P_i sent. Since $3d+1\le n$, then $n-2d\ge d+1$, which implies that in each subset of $n-2d$ there is at least one honest party. This means that by sending incorrect shares, the corrupted parties can only cause pair-wise inconsistency, which will result with a semi-corrupt pair. Thus, if all shares are consistent, the parties will hold the correct msg^i. Then, the parties can compare it to the value obtained in the first step and see whether P_i have cheated or not. We thus obtain the following protocol for Step 2:

(1) For each $i\in U$:
   (a) The parties locally compute $[\![\mathsf{msg}^i]\!]_{2d}$ via Eq. (1).
   (b) For each $j\in[n]$, the parties in each subset T where $|T| = n-2\cdot d$ and $P_j\notin T$ send their shares of msg^i to P_j.[6]
(2) If party P_j received contradicting shares in any of the n executions in the previous step, then it sets $\mathsf{cons}^j=1$. Otherwise, it sets $\mathsf{cons}^j=0$. Then, it broadcasts cons^j to the other parties.
(3) Upon receiving cons^j from all the parties:
   - If $\forall j:\mathsf{cons}^j=0$:
     The parties reconstruct msg^i for each $i\in[n]$. Let msg'^i be the compressed message of P_i agreed upon in the first step.
     – If $\forall i:\mathsf{msg}'^i=\mathsf{msg}^i$: the parties output accept.
     – If $\exists i:\mathsf{msg}'^i\neq\mathsf{msg}^i$: Let i be the *largest* such that $\mathsf{msg}'^i\neq\mathsf{msg}^i$, and let j the smallest index of party such that $i\neq j$. Then, the parties output reject, (i,j)[7]
   - If $\exists j:\mathsf{cons}^j=1$:
     Let j be the smallest index for which $\mathsf{cons}^j=1$. Let P_u and P_w be the first pair of parties who sent contradicting shares v_T^u and v_T^w to P_j, with T being the first subset for which $P_u,P_w\in T$ and $v_T^u\neq v_T^w$, and let $i\in[n]$ be the index of the execution in Step 1 where the inconsistency occurred. Then:
     (a) Party P_j broadcasts (T,i,u,w,v_T^u,v_T^w).
     (b) Party P_u broadcasts $\tilde{v}_T^u$ and party P_w broadcasts $\tilde{v}_T^w$.
     (c) If $\tilde{v}_T^u\neq\tilde{v}_T^w$, then the parties output reject, (u,w).
     Otherwise, if $v_T^u\neq\tilde{v}_T^u$, then the parties output reject, (j,u).
     Otherwise, the parties output reject, (j,w).

Cheating probability. The only way that the protocol can end with the parties outputting accept even though a corrupted party has cheated in the multiplication protocol, is if a random linear combination of incorrect messages yields the same value as a random linear combination of the correct messages. Using Lemma 1, we see that this probability is bounded by $\frac{1}{\omega_R}$. Thus, to obtain a statistical security of s bits, the parties will repeat the above protocol $\lceil\frac{s}{\log\omega_R}\rceil$ times.

Security. The only security concern in the above protocol, is that something can be learned from the additive shares of $(x\cdot y-r)^i$. This is prevented by randomizing the sharing when adding $[\![0]\!]_{2d}$. Formally, we define the ideal functionality $\mathcal{F}_{\mathsf{checkTrans}}$ in Functionality 13, which receives the sent/received messages from all parties, and the inputs and randomness of the honest parties. The latter suffices, in our setting of two-thirds honest majority, to compute all the messages that corrupted parties should have sent in the protocol. Thus, $\mathcal{F}_{\mathsf{checkTrans}}$ can find whether any party have cheated and sent incorrect messages. In case of cheating, it asks the real-world adversary to provide a pair of parties, such that at least one of them is corrupted, which is then output to the honest parties. Note that also in the case that no one cheated in the multiplication protocol, $\mathcal{S}$ is allowed to change the output to reject, but then it must provide also a semi-corrupt pair to eliminate. This captures the case when the corrupted parties send incorrect shares in the second step of our protocol, causing the verification to fail.

To simulate the protocol, note that by the definition of $\mathcal{F}_{\mathsf{checkTrans}}$, the simulator $\mathcal{S}$ receives from the trusted party computing $\mathcal{F}_{\mathsf{checkTrans}}$ all the corrupted parties' inputs and randomness, as well as all the messages sent/received in the protocol by the honest parties. Thus, $\mathcal{S}$ can perfectly simulate the first step. In the second step, it can perfectly simulate the messages sent when verifying msg^i for all i such that P_i is corrupted. This holds since all messages are a function of P_i's inputs and randomness which are known to $\mathcal{S}$, and shares distributed by $\mathcal{F}_{\text{zeroShare}}$, which is played by $\mathcal{S}$. Finally, when simulating the opening of msg^i for an honest P_i, the simulator $\mathcal{S}$ chooses random shares for the subsets containing honest parties only, under the constraint that all shares will open to msg^i and given the corrupted parties' shares which are known to $\mathcal{S}$. Since for

[6] This can be optimized by asking only one party in each set T to send the share to P_j and the rest send hashes of their shares for all $i\in[n]$.

[7] Note that here we know that P_i is corrupted and we could essentially remove only him. Nevertheless, since removing a pair of parties (with one of them being corrupt) maintains the threshold while reducing the overall communication, this is preferable.

FUNCTIONALITY 13 ($\mathcal{F}_{\text{checkTrans}}$- Verification of messages with Cheating Identification).
Let $\mathcal{S}$ be the ideal-world adversary controlling a subset $< n/3$ of corrupted parties.

(1) $\mathcal{F}_{\text{checkTrans}}$ receives from the honest parties their inputs, randomness and sent/received messages. These are used to compute the inputs and randomness of the corrupted parties.
(2) $\mathcal{F}_{\text{checkTrans}}$ sends $\mathcal{S}$ the corrupted parties' inputs and randomness and all messages the honest parties claimed to send/receive.
(3) Upon receiving from $\mathcal{S}$ all messages the corrupted parties claim to send/receive to/from honest parties:
   (a) If there is a contradiction between the message a corrupted party P_i claim to send/receive to/from an honest P_j, then $\mathcal{F}_{\text{checkTrans}}$ sends reject, (i, j) to the honest parties.
   (b) Otherwise, $\mathcal{F}_{\text{checkTrans}}$ checks that all messages sent from corrupted parties are correct given their inputs and randomness. If it holds, then $\mathcal{F}_{\text{checkTrans}}$ sends accept to $\mathcal{S}$. Otherwise, $\mathcal{F}_{\text{checkTrans}}$ sends reject to $\mathcal{S}$.
      - In the former case, $\mathcal{S}$ send back either accept or reject, (i, j) to the $\mathcal{F}_{\text{checkTrans}}$, such that either P_i or P_j (or both) are corrupt. This is then handed to the honest parties.
      - In the latter case, $\mathcal{S}$ must send back a pair (i, j) such that either P_i or P_j (or both) are corrupt. Then, $\mathcal{F}_{\text{checkTrans}}$ sends reject, (i, j) to the honest parties.

honest parties it holds that $\text{msg}'^i = \text{msg}^i$, and since all shares are randomized before opening, it holds that the honest parties' shares in the simulation are indistinguishable from their shares in the real execution. The only difference between the simulation and real execution is the case when $\mathcal{F}_{\text{checkTrans}}$ decides to reject, and the honest parties in the simulation accept. This happens when there is an incorrect message which is not detected since the random linear combination yields the same result as if correct messages were sent. As we have seen above, this can happen with probability 2^{-s} when repeating the process sufficient number of times.

We thus obtain the following:

Lemma 14. *Our protocol, as described in the text above, computationally computes $\mathcal{F}_{\text{checkTrans}}$ in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{zeroShare}})$-hybrid model, in the presence of malicious adversaries controlling d parties, where $n > 3d$, with statistical error 2^{-s}.*

Communication cost. Assume that the elements in R are represented using ℓ bits. In the first step, each party broadcasts, on average, 2 messages, and so 2ℓ bits. In the second step, each party sends $(2d+1) \cdot \binom{n-1}{2d} \cdot 2d$ elements to the other parties and broadcasts one bit, but if only one party in each set sends the shares and the other parties send a single hash, the factor $2d+1$ can be removed, asymptotically. With this in mind, overall, each party sends approximately $\binom{n-1}{2d} \cdot 2d \cdot \ell + 2\ell \cdot |BC|$ bits, where $|BC|$ is the cost of broadcasting one bit per party. When repeating the protocol s times, the above is multiplied by s. Note that the cost is completely independent from the number of verified multiplications.

4.3 Parties' Elimination and Recovery

In the previous section, we showed how to find a semi-corrupt pair. Once such a pair is found, the parties need to remove this pair from the computation in a secure way. After removing two parties where at least one is corrupt the new set of parties contains $n' = n-2$ participants, with $d' = d-1$ being corrupt. This new set of parameters preserves the required bound $d' < n'/3$, since $d < n/3$. Removing a semi-corrupt pair is fairly standard technique in Shamir secret-sharing-based protocols (e.g. [7]), and it can be extended to replicated secret-sharing in a reasonably straightforward manner, which we discuss in detail below.

Removing two parties, from which one is guaranteed to be corrupt, requires the parties to move from a d-out-of-n secret sharing to a $(d-1)$-out-of-$(n-2)$ secret sharing of each value on the output layer of the previous segment (which is the last state that was verified and approved by the parties).

For simplicity let us assume that all the values on the output layer of the previous segment are outputs of multiplication gates (otherwise, they are a linear function of multiplication's outputs). Recall that in our DN-style multiplication protocol, the output is computed by taking $[\![r]\!]_d + (xy - r)$. Recall also that this step is carried-out by having one subset of $n-d$ parties receive $xy - r$ from P_1 and add it to their share of r. To successfully eliminate a semi-corrupt pair, the parties thus need to prepare $[\![r]\!]_{d-1}$ and then let one subset of $n-2-(d-1) = n-d-1$ parties, that do not contain the two eliminated parties, add $xy-r$ to their share of r. To achieve this, we leverage the fact that $xy-r$, as well as each share of r, is held by a subset of $n-d \geq 2d+1$ parties, which means that among the $n-d$ parties there exists an honest majority.

Assume that P_i and P_j are the parties to eliminate. The parties work as follows:

(1) **Random keys update:** For each $i \in [n]$: for each subset $T \in \mathcal{P}$ such that $|T| = n-d$ and $P_i, P_j \in T$, the parties in T send the key $k_{i,T}$ to some party $P_u \notin T$. If P_u received different value for each key, then it chooses the majority value.
   Then, Party P_u uses these keys to compute r_T and adds it to its shares of r.
(2) Let $\mathcal{P}' = \mathcal{P} \setminus \{P_i, P_j\}$, $n' = n-2$ and $d' = d-1$. Note that after the previous step, each share of r is known to a set of active parties of size at least $n' - d' = n-2-(d-1) = n-d-1$.
(3) **From $[\![r]\!]_d$ to $[\![r]\!]_{d-1}$:**
   - For each subset $S \in \mathcal{P}'$ such that $|S| = n' - d'$, the parties in S initialize $r_S := 0$.
   - For each subset $T \in \mathcal{P}$ of size $n-d$:
     - *Case 1*: $P_i, P_j \notin T$.
       Note that there are $n-d$ subsets of size $n'-d'$ in each T (since $n'-d' = n-d-1$ and $\binom{n-d}{n-d-1} = n-d$). Then:
       The subset S containing the first $n'-d'$ parties in T sets $r_S \leftarrow r_S + (n-d) \cdot r_T$. The other subsets S of size $n'-d'$ parties set their share to be $r_S \leftarrow r_S - r_T$.
     - *Case 2*: $P_i \in T \wedge P_j \notin T$ or $P_j \in T \wedge P_i \notin T$.
       The subset $S = T \setminus \{P_i, P_j\}$ of size $n'-d'$ sets it share to be $r_S \leftarrow r_S + r_T$.
     - *Case 3*: $P_i, P_j \in T$.
       Let $S = T \setminus \{P_i, P_j\} \cup \{P_u\}$, where P_u is the party who learned r_T in the first step. Then, the parties in S set: $r_S \leftarrow r_S + r_T$.
(4) **Updating a multiplication's output: from $[\![z]\!]_d$ to $[\![z]\!]_{d-1}$.**
   Let $T \in P$ be the set of parties holding $x \cdot y - r$. Then:
   - *Case 1*: $P_i, P_j \notin T$.
     Let S be the set of the first $n'-d'$ parties in T. Then, the

parties in S set: $z_S = r_S + xy - r$. For each subset $S' \in \mathcal{P}'$ of size $n' - d'$ with $S' \neq S$, the parties in S set $z_S = r_S$.

- *Case 2*: $P_i \in T \wedge P_j \notin T$ or $P_j \in T \wedge P_i \notin T$.
  Note that $S = T \setminus \{P_i, P_j\}$ is a set of size $n' - d'$. Thus, the parties in S set $z_S = r_S + xy - r$, whereas for each subset $S' \in \mathcal{P}'$ of size $n' - d'$ with $S' \neq S$, the parties in S' set $z_S = r_S$.
- *Case 3*: $P_i, P_j \in T$.
  The parties in T send $xy - r$ to some party $P_u \notin T$. If P_u receives different values, then it chooses the majority value. Then, the parties in $S = T \setminus \{P_i, P_j\} \cup \{P_u\}$ set $z_S = r_S + xy - r$, whereas for each subset $S' \in \mathcal{P}'$ of size $n' - d'$ with $S' \neq S$, the parties in S' set $z_S = r_S$.

Communication cost. The above protocol requires interaction in two steps. First, the parties need to send all keys that are known by both P_i and P_j. Note that this cost is constant and does not depend on the size of the circuit. A second source of interaction is the case where both P_i and P_j are in the set of the parties who hold $xy - r$ for an output wire on the output layer of the last segment. Here we have $n - d$ elements that are sent for each output wire. Per party, the communication cost is thus bounded by $\frac{n-d}{n} \cdot W < W$, where W denotes the "width" of the circuit, i.e., the maximal number of multiplication gates that are on the same layer of the circuit. Note that W is *always* smaller than the size of the circuit, and for any "natural" circuit is of sublinear size.

5 SECURELY COMPUTING ANY FUNCTIONALITY OVER RINGS

We are now ready to present our main protocol to compute arithmetic circuits over the finite ring R. As explained before, the circuit is divided into segments, and each segment is computed separately. That is, the parties compute the segment using our private multiplication protocol, and then call $\mathcal{F}_{\mathsf{checkTrans}}$ to verify the correctness of the computation. If the parties receive accept from $\mathcal{F}_{\mathsf{checkTrans}}$, then they know that the secrets shared on the output layer of the current segment are correct, and so they can proceed to the next segment. Otherwise, they receive a semi-corrupt pair from $\mathcal{F}_{\mathsf{checkTrans}}$ which is removed from the computation. This is carried out by updating the secret sharing of the inputs to the current segment using our elimination and recovery subprotocol. The segment is then recomputed with less parties. More formally:

Input sharing step. At the end of this step, the parties will hold a consistent d-out-of-n secret sharing of each input.

(1) The parties set $n' = n$ and $d = t$, where $n = 3t + 1$.
(2) For each $i \in [n]$: party P_i distributes $[\![k_i]\!]_d$ to the other parties (the parties run a pairwise inconsistency check for $[\![k_i]\!]_d$. For each share that is inconsistent, P_i broadcasts the share to the parties). Note that this step is carried-out once, and $[\![k_i]\!]_d$ can be used to many computations.
(3) For the jth input x_j held by P_i: the parties locally derive $[\![r^i_j]\!]_d$ from $[\![k_i]\!]_d$ as shown in Section 3.2. Then, P_i broadcasts $x_j - r^i_j$ to the parties. Finally, the parties locally compute $[\![x_j]\!]_d = [\![r^i_j]\!] + x_j - r^i_j$.

Computing the next segment. This step begins with the parties holding a consistent secret sharing of the values on the input wires of the segment.

(4) The parties compute the segment gate after gate in some predetermined topological order. Linear gates are computed locally and multiplication gates are computed using Protocol 9.
(5) The parties send their inputs, randomness and sent/received messages in the execution of all multiplication protocols in the previous step to $\mathcal{F}_{\mathsf{checkTrans}}$. If $\mathcal{F}_{\mathsf{checkTrans}}$ sent accept, then they proceed to the next segment. Otherwise, they receive from $\mathcal{F}_{\mathsf{checkTrans}}$ a pair of parties (P_i, P_j) to eliminate. The parties then run the elimination and recovery subprotocol, set $n' = n' - 2$, $d = d - 1$ and go back to Step 4.

Output reconstruction. At the beginning of this step, the parties hold a d-out-of-n' secret sharing of each output. Then, for each output o_k intended to party P_i, the parties run reconstruct$([\![o_k]\!]_d, i)$.

We thus have the following (the proof is in the full version):

Theorem 15. *Let R be a finite ring, let f be a n-party functionality represented by an arithmetic circuit over R, and let $t \in \mathbb{N}$ be a security threshold parameter such that $n = 3t + 1$. Then, our main protocol, as described in the text, computationally computes f in the $(\mathcal{F}_{\mathsf{corRand}}, \mathcal{F}_{\mathsf{checkTrans}})$-hybrid model, in the presence of malicious adversaries controlling up to t parties.*

6 PERFORMANCE STUDY

In this section we study the concrete performance of our protocol. Our goal is to illustrate that, by means of our novel techniques, replicated secret-sharing can be used with reasonable efficiency for more than 3 or 4 parties, which are the traditional settings in which this scheme has been used. To this end, we provide an assessment of the communication and storage requirements of our protocol, for different parameters, in Section 6.1. In addition, we completely[8] implemented our protocol in C++, and in Section 6.2 we discuss in detail the experimental results we have obtained. The source code of our implementation can be found in https://github.com/anderspkd/ccs-DEN22.git.

6.1 Storage and Communication Costs

Communication costs. We begin by deriving an expression for the communication complexity of our protocol from Section 5. Let $|C|$ be the size of circuit (measured by the number of multiplication gates). Let $|S|$ be the size of a segment, and let $m = |C|/|S|$ be the number of segments. Recall that the cost of our multiplication protocol is $1 + \frac{d-1}{n}$ ring elements per party, which we upper-bound by $4/3$. Furthermore, let Ch_n be the cost of the multiplication check with n parties, which equals $\binom{n-1}{2t} \cdot 2t$ ring elements, plus the cost of broadcasting two ring elements (recall this overall cost is exponential in n, but independent of the size of the segment). The communication cost of *one segment* in our protocol is $\frac{4}{3} \cdot |S| + \mathsf{Ch}_n$ ring elements per party. In the optimistic case, where all parties act honestly, there are $m = |C|/|S|$ segments executed, which leads to a communication complexity of $\frac{4}{3}|C| + m \cdot \mathsf{Ch}_n$ ring elements per party.

[8] Except for some minor steps that are not expected to affect runtimes drastically.

n	Share size	Communication	
		Mult. gate (opt/worst)	Checks (opt/worst)
4	24 B	8 B/16 B	48 B/48 B
7	120 B	9.14 B/27.43 B	480 B/528 B
10	672 B	9.6 B/38.4 B	3.94 KiB/4.45 KiB
13	3.87 KiB	9.85 B/49.23 B	30.94 KiB/35.39 KiB
16	23.46 KiB	10 B/60 B	234.61 KiB/270 KiB
19	145.03 KiB	10.11 B/70.74 B	1.7 MiB/1.96 MiB
22	908.44 KiB	10.18 B/81.45 B	12.42 MiB/14.38 MiB
25	5.61 MiB	10.24 B/92.16 B	89.78 MiB/104.16 MiB
28	35.76 MiB	10.29 B/102.86 B	643.64 MiB/747.8 MiB
31	229.23 MiB	10.32 B/113.55 B	4.48 GiB/5.21 GiB
n	$\binom{n-1}{t} \cdot \ell$	$\ell \cdot (1 + \frac{t-1}{n}) \Big/ \times(1+t)$	$\mathsf{Ch}_n \Big/ + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$

Table 1: Storage and communication complexity (per party) of our protocol for different number of parties $n = 3t + 1$. The bit-size ℓ of each ring element is assumed to be 64 bits. $|BC|$, the cost of broadcasting one bit per party, is taken to be 0 as its contribution is minimal, and the number of repetitions of the check to achieve negligible soundness is assumed to be 1. We consider the case in which the whole circuit is one single segment, so only one check is performed at the end of the execution. We report complexities for the optimistic case where no cheating occurs, and for the worst case where the adversary repeats the circuit t times (each time with three parties less). This causes a multiplicative overhead of $(1 + t)$ in the complexity per multiplication gate, and an additive overhead of $\sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$ in the complexity regarding the checks. Here $\mathsf{Ch}_n = \binom{n-1}{2t} \cdot 2t \cdot \ell + 2\ell \cdot |BC|$.

In the case of active cheating, several segments might be executed multiple times. The exact communication complexity in this case depends heavily on where the adversary cheats, and how many times he does so. However, in terms of the worst case it is easy to see that the scenario that leads to the most expensive communication complexity is when the adversary behaves honestly for all segments except for the last one, point in which the adversary misbehaves, making this segment be executed t more times, reducing the number of parties by three in each repetition.[9] As a result, the worst case communication complexity in the event of active cheating by adding, to the optimistic case above, the cost of the t extra repetitions, which is given by $\frac{4}{3}|S|t + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}$.

This way, we can write the worst case complexity per party as

$$\frac{4}{3}|C|(1 + \frac{t}{m}) + m \cdot \mathsf{Ch}_n + \sum_{\ell=1}^{t} \mathsf{Ch}_{n-3\ell}.$$

We discuss possible choices of m below.

Concrete complexity for certain parameters. As we have mentioned, certain metrics of our protocol increase exponentially with the number of parties. For instance, the communication complexity of the final check, although is independent of the number of multiplications being checked, is exponential in n. In addition, storage complexity, which is directly related to the size of each share, is exponential in n, and this also affects computation involving shares, like locally adding secret-shared values or reconstructing a secret from a given set of shares. However, we recall that a crucial aspect of our protocol is that its communication complexity, apart from the final check, does not grow exponentially with the number of parties, and in fact it is kept constant (per-party); this is in contrast to many existing work that makes use of replicated secret-sharing.

In Table 1 we see the share size, together with the communication cost of each multiplication gate and the final check, per party, for a 64-bit ring and increasing number of parties. We consider one single segment corresponding to the whole circuit, meaning that $m = 1$, and there is only one single check at the end of the execution. We report complexities for both the optimistic case (when there is no cheating) and the worst-case (when the circuit is re-run t more times, each with 3 parties less).

In the optimistic case, the communication cost per multiplication gate (regardless of the number of parties) is kept under 11 bytes, while in the worst case when the circuit is evaluate t more times it goes up by a factor of $(1 + t)\times$. This multiplicative overhead takes a toll when n is moderately large, like for $n = 31$, where it increases the cost per gate from around 10 to 110 bytes, an overhead of 10×. Regarding the communication arising from the different checks, in the optimistic case only one check is executed, but in the worst case t more checks must be performed, each with three parties less. Fortunately, from Table 1 we see that this overhead, being *additive*, is quite small with respect to the check in the optimistic case, increasing from around 4.5 to 5.2 gigabytes for $n = 31$, for example.

We see that the communication complexity of the final check grows very fast, even when compared to the share size. However, we stress that this is only executed *once* at the end of the protocol. Depending on the application at hand, this overhead could be considered acceptable with respect to the rest of the computation. To illustrate this we study, for different number of parties, the number of multiplications needed so that the communication complexity involved in their computation *matches* the communication complexity of checking them, which means that the overhead of the final check at this point is 2x, and it approaches 1x as the number of multiplications grow.[10] For moderately large values of n such as $n = 10$, the check costs the same as less than one thousand multiplications, and for larger values like $n = 22$, the check costs the same as a bit over one million multiplication. This grows up to roughly one billion if $n = 31$. A detailed analysis for more values of n appears in the full version. Furthemore, other additional aspects of the communication complexity of our protocol appear in the full version.

Finally, the share size, which grows exponentially with n, is kept in the order of bytes for $n = 4, 7, 10$, kilobytes for $n = 13, 16, 19, 22$, and tens of megabytes for $n = 25, 28$. For $n = 31$, this size reaches around 200 megabytes, which is large when considering that this corresponds to *each* shared value. However, the following optimization can prove to be crucial for reducing the impact of this overhead

[9] The protocol from Section 5 is described as removing *two* parties in each repetition, but it is easy to see that the bound $t < n/3$ can be preserved while removing *three*, which helps efficiency and results in the four-party case being the base case.

[10] We remark that this only measures the amount of messages sent. We must take into account that the computation of the multiplication gates happens in several rounds, while the check only uses a constant number of rounds, which in practice makes it more efficient to compute even if its communication is the same (or even more) than evaluating several multiplications.

in practice. As currently described, our protocol requires the parties to store *all* the secret-shared values computed in a given segment, to be able to check them at the verification step. Instead, the parties can sample the random coefficients δ_i used to compress the values to be checked via a linear combination *on the fly*. In a bit more detail, after performing the computation of a given multiplication layer of the circuit (in particular, after the adversary committed to its errors) the parties sample the necessary random values for the given layer, and aggregate these into a small amount of secret-shared values that correspond to these computed in the final verification step. This way, the parties can discard the shares obtained in a given layer (unless they are required for a subsequent step in the computation). A similar approach was also used in the context of MPC with dynamic participants in [22], in order to reduce the number of shared values needed from one round to the next.

On the choice of m. For the results in Table 1 we have chosen $m = 1$, so we regarded the whole circuit as one single segment, and only one check is performed at the end. If this check fails then the entire circuit is re-run, with three parties less. Choosing $m = 1$ leads to the best possible total communication complexity in the optimistic case (which is arguably the scenario more relevant in practice), but the gap between the optimistic and worst-case scenarios is very large. If, instead, it is the goal to minimize this gap, we could take larger values of m. For example, $m = t$ leads to an overhead in the amortized communication complexity per multiplication gate in the worst-case of only $2\times$ with respect to the optimistic case, but now the latter is more expensive as the check is performed t times, instead of just one.

Different choices of m lead to different performance results, and which one is optimal depends on the expected "amount of cheating". We defer to the full version a more detailed discussion on some concrete values of m and their effect on the communication complexity.

6.2 Experimental Results

We created a proof of concept implementation of the core parts of our contribution, namely the multiplication and check protocols. Our intention is to investigate, in concrete terms, the overhead of our techniques with a varying number of parties and computation sizes. Our implementation can be found alongside this submission, as can all the experimental data we generated and analyze in this section. To the best of our knowledge, our implementation and evaluation constitute the first set of experimental results regarding replicated secret-sharing for an increasing number of parties.

Our implementation was written in C++ and all experiments where run on a single `c5.4xlarge` AWS instance, with each party being executed in a different procedure. We set our experiments in a study a WAN by setting a delay of 100ms and a bandwidth of 100 Mpb/s. We believe that this creates an experimental setup that is easier to replicate. We choose a prime field of approximately 64 bits (so in particular, the final check only needs to be repeated once for a statistical security of $\approx 2^{-64}$).

6.2.1 Experiments. Recall that, when the number of parties increases, storage, local computation and the communication of the final check increase *exponentially*, but the communication complexity per multiplication gate remains *constant*. As a result, one might expect our protocol to *not* be very competitive in scenarios such as the following:

- The number of parties is large (so storage and local computation becomes very expensive), *and*
- The network is reasonably fast[11] (so the fact that the communication per multiplication is small does not provide a benefit with respect to the first item), *and*
- The circuit is relatively small (so the benefit of the complexity of the final check being independent of the circuit size matters less).

However, our protocol can potentially become competitive if any of these conditions does not hold. The goal of our experiments is to study precisely this hypothesis, that is, in which settings the exponential nature of our protocol represents an insurmountable overhead, and which cases our protocol can prove beneficial.

We run several experiments to investigate the computation and communication complexity. While the communication complexity has already been analyzed in the previous section, the computation complexity has not. In particular, the use of replicated secret-sharing imposes a non-trivial computationally overhead due to all of the combinatorics involved. Thus the goal is to shed some light on how this complexity grows with the number of inputs and parties, respectively.

For both multiplication and check protocols, we run several experiments where we vary the number of inputs and parties, respectively. We expect to see that *local* computation matters more, the higher the number of parties, and that this complexity outweighs the communication complexity.

6.2.2 Multiplication. Our first experiment perform a number of multiplications, where we measure the time to compute the product $\odot$ between the shares; the time it takes to send around shares, and the time it takes to receive and adjust them. Results can be seen in Figure 1. What is clearly, visible, is that local computation quickly becomes dominant, as the number of multiplication grows and parties grows.

6.2.3 Check. Our second experiment, executes the check protocol on a variable number of multiplications. The results can be found in Figure 2. As expected, we see that the time is essentially constant, regardless of the number of multiplcations. (The outlier for $n = 4, 7$ can be explained by variability in the experimental data, and we believe it should dissapear by increasing the number of trials.) However, we also see that local computation (represented by the blue bar) increases in significance as both the number of parties and the number of things to check, increases.

6.2.4 End-to-End. Finally, for the sake of completeness we include some end-to-end results. As we have already argued, our main goal is to explore the concrete practicality of replicated secret-sharing-based protocols for increasing number of parties, for which the experiments presented in previous section are more useful since they show the relative performance of the different parts of our protocol, and they allow us to see how the exponential blow-up

[11] This is partly our motivation for choosing a (simulation of a) WAN network for our experiments.

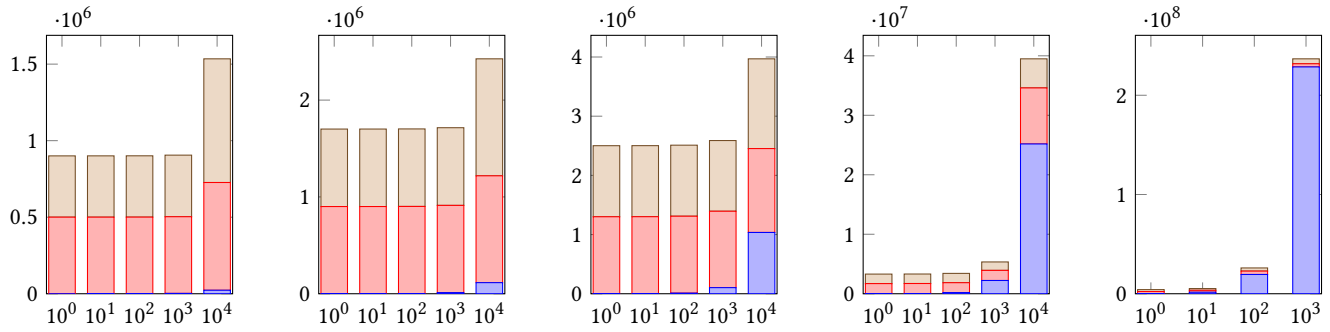

Figure 1: Multiplication protocol. Each graph represents a different choice for the number of parties $n \in \{4, 7, 10, 13, 16\}$ while each bar in each graph represents the number of multiplications performed (x-axis). The y-axis represents time in microseconds. In each graph the blue section corresponds to the local product each party performs, the red section is the time it takes to send these messages to P_1, and the beige section is the time it takes for P_1 to send the reconstructions back.

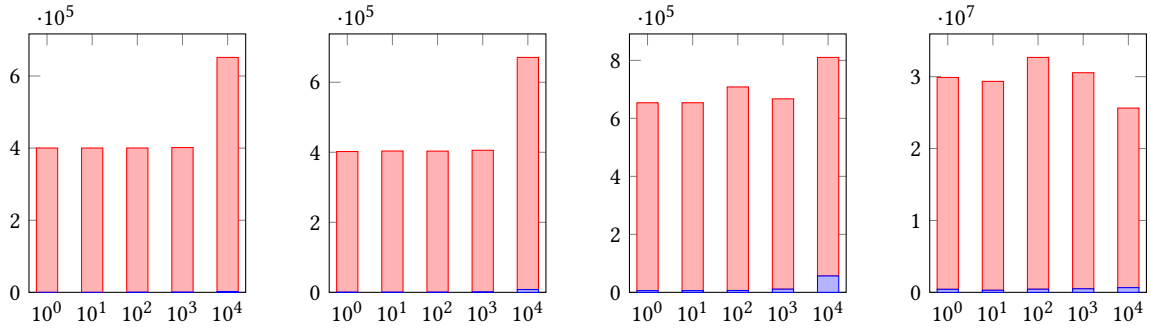

Figure 2: Check phase. Blue represents preparing $[\![\text{msg}^i]\!]_{2d}$, while the red represents reconstruction. Each graph corresponds to $n \in \{4, 7, 10, 13\}$, the x-axis is number of multiplications that is checked, while the y-axis is time in microseconds.

n	Input phase (s)	Mult. phase (s)	Check phase (s)
4	0.6	1.5	0.6
7	1.2	2.4	0.6
10	1.8	3.9	0.8
13	2.5	39.5	25.6

Table 2: End-to-end runtimes (in seconds) for a circuit with 100 input gates and 10000 multiplication gates distributed across one layer, for a varying number of parties.

of our protocol manifests itself in practice. However, we believe it is fruitful to consider end-to-end runtimes as a *rough* estimate of how our protocol would fare in certain tasks. We warn, however, that these numbers are highly volatile as they strongly depend on the experimental setting, quality of implementation, etc., and they should only be used as a coarse guideline.

We consider a circuit with 100 input gates, and 10000 multiplication gates distributed across one layer. Our results are presented in Table 2.

6.2.5 Discussion. The results of the experiments we performed seem to be in accordance with our hypothesis. Specifically, local computation quickly becomes dominant, in particular in the multiplication protocol, and so our protocol would, in those cases, benefit more from a slower network.

This is in particular relevant, taking into consideration low communication cost of the check, as pointed out earlier.

ACKNOWLEDGMENTS

A. Nof supported by ERC Project NTSC (742754). This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2021 JP Morgan Chase & Co. All rights reserved.

REFERENCES

[1] Emmanuel Abbe, Amir E. Khandani, and Andrew W. Lo. 2012. Privacy-preserving methods for sharing financial risk exposures. *American Economic Review* 102, 3 (2012), 65–70.

[2] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. 2019. Efficient information-theoretic secure multiparty computation over Z/pk Z via Galois rings. In *Theory of Cryptography Conference*. Springer, 471–501.

[3] Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. 2021. An efficient passive-to-active compiler for honest-majority MPC over rings. In *ACNS*.

[4] Alessandro N. Baccarini, Marina Blanton, and Chen Yuan. 2020. Multi-Party Replicated Secret Sharing over a Ring with Applications to Privacy-Preserving Machine Learning. *IACR Cryptol. ePrint Arch.* (2020).

[5] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants. In *ACM CCS*.

[6] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-Secure MPC with Linear Communication Complexity. In *TCC*.

[7] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-Secure MPC with Linear Communication Complexity. In *TCC*.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *ACM Symposium on Theory of Computing*.

[9] Bonnie Berger, Cho Hyunghoon, and David J. Wu. 2018. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology* 36, 6 (2018), 547.

[10] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. 1992. Bit optimal distributed consensus. In *Computer science*. 313–321.

[11] Dan Bogdanov, Riivo Talviste, and Jan Willemson. 2012. Deploying Secure Multi-Party Computation for Financial Data Analysis - (Short Paper). In *Financial Cryptography and Data Security FC*.

[12] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. 2009. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security FC*.

[13] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*.

[14] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*.

[15] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *ACM CCS*.

[16] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2020. Efficient Fully Secure Computation via Distributed Zero-Knowledge Proofs. In *ASIACRYPT*.

[17] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology* 13, 1 (2000), 143–202.

[18] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *ACM STOC*.

[19] Erika Check Hayden. 2015. Extreme cryptography paves way to personalized medicine. *Nature News* 519, 7544 (2015), 400.

[20] Jung Hee Cheon, Dongwoo Kim, and Keewoo Lee. 2021. MHz2k: MPC from HE over $\mathbb {Z}_{2^k}$ with New Packing, Simpler Reshare, and Better ZKP. In *CRYPTO*.

[21] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO*.

[22] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2021. Fluid MPC: Secure Multiparty Computation with Dynamic Participants. In *CRYPTO*.

[23] Brian A. Coan and Jennifer L. Welch. 1989. Modular Construction of Nearly Optimal Byzantine Agreement Protocols. In *ACM Symposium on Principles of Distributed Computing*.

[24] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPDZ2k: Efficient MPC mod 2k for Dishonest Majority. In *CRYPTO*.

[25] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC*.

[26] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. 2003. Efficient Multi-party Computation over Rings. In *EUROCRYPT*.

[27] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2020. Secure Evaluation of Quantized Neural Networks. *Proc. Priv. Enhancing Technol.* 2020, 4 (2020), 355–375.

[28] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX*.

[29] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *IEEE Symposium on Security and Privacy, SP*.

[30] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO*.

[31] Danny Dolev and H. Raymond Strong. 1983. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666.

[32] Daniel Escudero and Eduardo Soria-Vazquez. 2021. Efficient Information-Theoretic Multi-party Computation over Non-commutative Rings. In *CRYPTO*.

[33] Jun Furukawa and Yehuda Lindell. 2019. Two-Thirds Honest-Majority MPC for Malicious Adversaries at Almost the Cost of Semi-Honest. In *ACM CCS*.

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

[35] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM STOC*.

[36] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. 2018. Secure Computation with Low Communication from Cross-Checking. In *ASIACRYPT*.

[37] Vipul Goyal, Yanyi Liu, and Yifan Song. 2019. Communication-Efficient Unconditional MPC with Guaranteed Output Delivery. In *CRYPTO*.

[38] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *CRYPTO*.

[39] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. 2016. High-Precision Secure Computation of Satellite Collision Probabilities. In *SCN*.

[40] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan* (1989).

[41] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX*.

[42] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2021. Tetrad: Actively Secure 4PC for Secure Training and Inference. Cryptology ePrint Archive, Report 2021/755. https://ia.cr/2021/755. To appear at NDSS 2022.

[43] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. 2010. Information-Theoretically Secure Protocols and Security under Composition. *SIAM J. Comput.* 39, 5 (2010), 2090–2112.

[44] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. 2020. Overdrive2k: Efficient Secure MPC over $\mathbb {Z}_{2^k}$ from Somewhat Homomorphic Encryption. In *CT-RSA*.

[45] Tal Rabin and Michael Ben-Or. 1989. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *ACM Symposium on Theory of Computing*.

[46] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *Symposium on Foundations of Computer Science*.