Scalable Secure Multiparty Computation

Ivan Damgård[1*] and Yuval Ishai[2**]

[1] Aarhus University (ivan@daimi.au.dk)
[2] Technion (yuvali@cs.technion.ac.il)

Abstract. We present the first general protocol for secure multiparty computation which is *scalable*, in the sense that the amortized work per player does not grow, and in some natural settings even vanishes, with the number of players. Our protocol is secure against an *active* adversary which may *adaptively* corrupt up to some *constant fraction* of the players. The protocol can be implemented in a constant number rounds assuming the existence of a "computationally simple" pseudorandom generator, or in a small non-constant number of rounds assuming an arbitrary pseudorandom generator.

1 Introduction

We consider the complexity of general secure multiparty computation (MPC) [37, 23, 9, 12] involving a large number of players. This problem is motivated by real-life applications that require a large-scale cooperation between many mutually distrustful entities. Alternatively, one may wish to distribute a critical role of a single "dealer" (distributing keys or other forms of correlated randomness) among many untrusted players.

As another motivating example, consider the case where a *small* number of players wish to securely perform some complex distributed computation on their inputs. In this case, the players could directly interact with each other using known methods from the MPC literature. However, all these methods have some inherent weaknesses. The first type of methods (for MPC with honest majority) will completely fail if a majority of the players become corrupted. Even if players trust each others' "honorable intentions", they may still be subject to hacking and other external attacks. Since attacking successfully a small number of players is sufficient to corrupt a majority, one may need to invest heavily in physical and other security measures to make this solution work. The second type of methods (for MPC with no honest majority) cannot provide important security features such as fairness or guaranteed output delivery [13]. Moreover, protocols of the latter type make use of expensive zero-knowledge proofs and public-key primitives.

* Supported by BRICS, Basic research in Computer Science, Center of the Danish National Research Foundation and FICS, Foundations in Cryptography and Security, funded by the Danish Natural Sciences Research Council.

** Supported by grant 2004361 from the U.S.-Israel Binational Science Foundation.

One could hope to avoid the weaknesses discussed above by distributing the computation among a *large* number of players with diverse platforms and security measures. The event that a substantial fraction, say 25%, of this large number of players become corrupted may be considered much less likely than the adversary breaking into a majority of the original small set of players.

For all of the application scenarios described above, one could in principle employ any of a large variety of general MPC protocols from the literature. However, a major drawback of all these protocols is that their efficiency *does not scale well* with the number of players. There has been a long line of work attempting to improve this state of affairs by reducing the communication complexity of general MPC protocols [20, 21, 25, 24, 29, 15, 17, 26, 8]. Still, even in the most efficient protocols to date [24, 26, 8], the amount of communication involving *each player* grows linearly with the number of other players.

1.1 Our Results

In this work, we show that this is not an inherent state of affairs by presenting the first *scalable* MPC protocol. In this protocol, the amortized work per player does not grow, and in some cases even vanishes, with the number of players, while still tolerating a constant fraction of maliciously corrupted players.

It is convenient to formulate our results in the clients-servers model of [14, 16]. This model refines the standard MPC model by separating between *clients*, who provide inputs and receive outputs, and *servers* who perform the actual computation. (Note that the same party can play both roles, as is the case in the standard model of secure computation.) The main advantage of this refinement is that it allows to decouple the number of "consumers" (clients) from the expected "level of security" (which depends on the number of servers and the security threshold). In all cases, we require security against a computationally-bounded adversary that may (actively, adaptively) corrupt at most some *constant fraction* of the servers and *any* strict subset of the clients.

Our main result comes in two flavors:

- In the standard MPC setting, where every player can hold an input and receive an output, the amount of work involving each player is $\tilde{O}(|C|)$, where $|C|$ is the size of the circuit(s) being evaluated. Here the $\tilde{O}(\cdot)$ notation ignores factors that depend polynomially on the security parameter k and polylogarithmically on the number of players,[3] as well as additive terms that depend on the number of players and the size of the inputs, but not on $|C|$ (see more below). The security of this protocol can be based either on the existence of a

[3] Since the security parameter needs to be (at least) polylogarithmic in the number of players, the polylog(n) term can be viewed as being absorbed in the poly(k) term. Also, the polylog(n) term only applies to the computation and not to the communication.

"computationally simple" pseudorandom generator (PRG)[4] (with a constant number of rounds) or on an arbitrary PRG (with poly(k) rounds).
- In the case where the number of clients is constant, as in the above motivating example, the *total* amount of work of all clients and servers is $\tilde{O}(|C|)$. Security in this case is based on the existence of a computationally simple PRG.

The $\tilde{O}(|C|)$ communication complexity of the latter protocol is essentially the best one could hope for given the current state of the art in the area of secure computation. Indeed, it is not known how to break this bound even in the case of security against a *single, semi-honest* player, with the only exceptions being protocols that apply to special classes of circuits (e.g., constant-depth circuits [4]) or protocols that involve an exponential amount of computation [6, 34]. Thus, even in a much more liberal setting than the one addressed here, significantly improving the $\tilde{O}(|C|)$ bound would be considered a major breakthrough.

The main disadvantage of our protocol compared to most previous works along this line is its non-optimal security threshold.[5] However, in our scenario we view the improved efficiency as being *qualitatively* more significant than the difference between different "constant" levels of resilience. The latter is very important in the case of computations involving *few* players. For instance, improving the resilience threshold from $t < n/3$ to $t < n/2$ enables three players to perform tasks that otherwise could not be performed at all. This difference is less significant in our scenario, where the number of players is large: in practice, it seems unlikely that increasing the number of players would automatically allow an adversary to corrupt more players. Therefore, the clients can compensate for the degraded level of security by employing a larger number of servers. Using our protocol, this can be done at a low extra cost.

A major advantage of our protocol over some previous related works (e.g., [15, 26]) is its provable *adaptive* security. Insisting on adaptive security is very significant when the number of players is large, since otherwise it would be possible to fool the (non-adaptive) adversary by assigning the computation to a small, randomly chosen, subset of the players. We note, however, that in contrast to the above simple approach, it is not clear if the protocols of [15, 26] are actually insecure in any "harmful" sense in the presence of an adaptive adversary.

On the amortized measure of complexity. We stress again that our asymptotic complexity measurements ignore *additive* terms that depend (polynomially) on the number of players and the security parameter, but do not depend on the size of the circuitry to be evaluated. Thus, the scalability of our protocol only holds in an *amortized* sense, i.e., when amortizing over (one or several) "large"

[4] Specifically, we assume the existence of a PRG in NC^1 or similar complexity classes. This is a mild assumption, which in particular is implied by virtually all standard *concrete* intractability assumptions used in cryptography (see [1] for discussion).

[5] We did not attempt to optimize the exact resilience of our protocol. In particular, in some cases we give away resilience in order to simplify subprotocols and their analysis. We note, however, that giving up on *some* resilience is inherent to our technique.

circuits whose total size is much larger than the number of servers. The latter scenario is very realistic even for one-time computations (as in the contexts of distributed data-mining on large databases, or distributed generation of cryptographic keys), let alone when the computation is used to generate resources for a life-time of future interactions. Similar conventions were used in previous works along this line [25, 15, 24, 17, 26]. Similarly to previous works, we also ignore the cost of broadcasting the inputs held by the input clients to the n servers. This cost is inevitable in the general case (consider the functionality which outputs to all players the concatenation of all inputs) and can be *entirely avoided* in the case of a small number of clients. Moreover, in many applications the inputs to the computation are either short or nonexistent (as in the case of emulating a trusted dealer of correlated randomness). Thus, the amortized complexity of our protocol is typically quite insensitive to whether a broadcast channel is given,[6] and in many natural scenarios reflects their actual cost.

1.2 Techniques

Our protocol employs a variety of previous tools and techniques, as well as new ones. One source for improvement over previous works in the area is the use of the "share-packing" technique of Franklin and Yung [21]. To employ this technique in our context we need to combine it with an efficient procedure for generating many packed random secrets that are guaranteed to satisfy some given (possibly complex) replication pattern. The latter procedure extends previous techniques of Hirt et al. [25, 24]. Another source for improvement is the use of the "easy PRG" technique of Applebaum et al. [2], which in turn relies on Yao's garbled circuit technique [37] together with information-theoretic randomization techniques [28]. These previous tools are described in Section 3.

At a very high level, our protocol proceeds roughly as follows. Each client concatenates to its inputs a vector of replicated random bits, i.e., certain pairs of these bits must be equal. The replication structure (that is, which pairs have to be equal) is derived from the circuit C being evaluated. This random input vector is of roughly the same size as C. The client then divides this string into blocks of size $\Theta(n)$, applies to each block an efficient (constant-rate) Reed-Solomon encoding, and sends to each server a single entry of the encoding of each block. Now the servers engage in an efficient interactive testing procedure, verifying that the encoded blocks distributed by each client are not far from being valid encodings of vectors that are consistent with the structure of C. This procedure may result in eliminating some of the blocks. Next, each server locally applies a *low-degree* computation to its local information and sends the result back to the clients. (This step does not depend on the structure of C.) Finally, the output of the protocol is obtained by each client by first applying error correction of Reed-Solomon codes and then applying a decoding procedure that essentially

[6] The constant rounds feature of our protocol can also be maintained (settling for *expected* constant rounds) without the availability of broadcast [19, 31, 30].

mimics the computation of C. The intuition on how the previous tools are used in the protocol will be explained towards the end of Section 3.

Related work. In a concurrent and independent work, Hirt and Nielsen [27] present a different MPC protocol whose (amortized) communication complexity per player is linear in the circuit size and independent of the number of other players. This work significantly differs from ours both in the features of the protocol and in the underlying techniques. The main advantage of the protocol from [27] is that it achieves an optimal security threshold ($t < n/2$). However, unlike our protocol, it cannot be proved to be adaptively secure. Moreover, it does not match the complexity of our protocol in the case of a constant number of clients (as in a distributed implementation of two-party computation). Recall that in the latter case, the *total* complexity of our protocol can be made linear in the circuit size. The two works differ also in the underlying cryptographic assumptions and primitives. The protocol of [27] employs a threshold homomorphic public-key encryption, whereas our protocol involves primitives from the "private-key" world. Thus, the current work and [27] are quite complementary.

Organization. Following some preliminaries (Section 2), in Section 3 we describe previous tools and techniques on which our protocol relies. In Section 4 we describe a scalable MPC protocol under some setup assumptions, which are eliminated in Section 5. For lack of space, some of the material was omitted from this extended abstract and can be found in the full version.

2 Preliminaries

We consider a system consisting of several players, who interact in synchronous rounds via authenticated secure point-to-point channels and a broadcast medium. (The availability of broadcast is not a necessary assumption, and the number of broadcasts employed by our protocols will be independent of the size of the circuit being evaluated.)

Players can be designated three different roles: *input clients* who hold inputs, *output clients* who receive outputs, and *servers* who may be involved in the actual computation. This is just a generalization of the standard model, where each player can play all three roles. We denote the number of servers by n. We will sometimes refer to a server or a client as a "player" and denote server i by P_i. The functionalities we wish to compute only receive inputs from input clients and only provide outputs to output clients. For simplicity we will only explicitly consider deterministic functionalities providing all output clients with the same output, though an extension of our results to the general case is straightforward. (In contrast, we will employ subprotocols that compute randomized functionalities and provide servers and input clients with outputs as well.)

We assume by default an active, adaptive, rushing adversary corrupting at most t servers, where t is bounded by some constant fraction of n that will be either explicitly specified or understood from the context. The adversary may also corrupt an arbitrary strict subset of the clients. We refer the reader to, e.g., [10] for the standard definition of security in this model. Our protocol heavily builds on previous work in the area of *information-theoretic* multiparty computation. We assume some familiarity of the reader with basic techniques in this area, such as Shamir's secret-sharing [36] and the "BGW protocol" [9].

We will use secret-sharing over a finite field K of characteristic 2, where $|K| = O(n)$. We let $\omega_1, \ldots, \omega_n$ denote n distinct elements of K, where ω_i will be an interpolation point assigned to server P_i.

The linear preprocessing model. It will be convenient to first describe and analyze our protocols in the *linear preprocessing model,* where we allow some restricted trusted setup as described below. We later show how the protocols can be converted to the *plain model*, where no setup assumptions are made.

In the linear preprocessing model, we assume a dealer who initially gives to each player a set of values in K or in its subfield $\mathrm{GF}(2)$. The values distributed by the dealer are restricted to be "linearly correlated". Specifically, the dealer picks a random codeword in a linear code defined over K or over $\mathrm{GF}(2)$, and then hands to each player a subset of the coordinates in the codeword. It is public which subsets are used, but the values themselves are private. This procedure can be repeated multiple times, possibly using different linear codes.

Of course, we do not expect that a dealer as above would exist in practice. This is only a convenient abstraction, that can be formalized as an ideal functionality. Much of the effort in the paper is devoted to finding efficient methods for amortizing the cost of multiple invocations of this functionality, for the types of distributed linear codes employed by our high level protocol.

A linear correlation pattern as above can be generally specified via a *distributed linear code* $L = (V, \pi_1, \ldots, \pi_n)$, where V is a linear code (vector space) over K or over $\mathrm{GF}(2)$, and π_i is the set of coordinates assigned to P_i. For $v \in V$ we use the notation $v(i)$ to denote v_{π_i}, the restriction of v to coordinates belonging to player i, which naturally extends to $v(H)$ for a set $H \subseteq [n]$ of players.

3 Basic Tools

In this section we present two tools from previous work on which our protocol relies: the share packing technique of Franklin and Yung [21] (Section 3.1) and the "easy PRG" technique of Applebaum et al. [2] (Section 3.2).

3.1 Share Packing

Most MPC protocols from the literature distribute the inputs and the random inputs via the use of Shamir's secret-sharing scheme [36]. This scheme is *ideal*, and in particular its share size is the best possible using any information-theoretic threshold scheme. However, viewed as an encoding of the shared secret, the rate achieved by Shamir's scheme is still very low: the size of the encoding is n times larger than the size of the secret. This suggests that better efficiency could be obtained via the use of better encodings.

A first and major step in this direction was taken by Franklin and Yung [21].[7] They observe that for the purpose of MPC, one does not necessarily have to rely on a strict threshold scheme in which every t players learn nothing about the secret and every $t+1$ players can fully reconstruct it. Instead, the latter requirement can be relaxed: it is only required that the encoding be "sufficiently robust". (Such relaxed secret-sharing schemes are known as *ramp schemes*.) Specifically, the following natural *multi-secret* generalization of Shamir's scheme is used. Let t be a security threshold and $\ell \geq 1$ be a parameter specifying a number of secrets to be jointly shared between the players. Assume that the field K has more than $\ell + n$ elements, and associate with each secret index $1 \leq i \leq \ell$ a distinct field element $\mu_i \in K$, different from all elements ω_j assigned to players. A block of ℓ secrets $(s_1, \ldots, s_\ell) \in K^\ell$ is distributed as follows.

Procedure $\mathsf{ShareBlock}(K, \ell, t, (s_1, \ldots, s_\ell))$:

- Pick a random polynomial p of degree (at most) $T = t+\ell-1$ over K, subject to the constraints $p(\mu_i) = s_i$, $1 \leq i \leq \ell$.
- Send to each player P_j the share $p(\omega_j)$.

It is easy to verify that every t players learn nothing about the secret block $\mathbf{s} = (s_1, \ldots, s_\ell)$ from the shares they receive in ShareBlock. Moreover, this type of secret-sharing supports the homomorphic properties required by MPC protocols. Indeed, suppose that two blocks of secrets $\mathbf{s}, \mathbf{s}'$ have been distributed using polynomials p, p', respectively. Then, by locally adding their shares the players hold points on the degree-T polynomial $p + p'$ whose value at point μ_i is $s_i + s'_i$. Thus, two shared blocks can be locally added. Similarly, by locally multiplying their shares the players hold points on the degree-$2T$ polynomial $p \cdot p'$ whose value at point μ_i is $s_i \cdot s'_i$. Given these properties, it was suggested in [21] to generalize the standard MPC protocols by applying them to packed shares instead of standard shares. This allows to simultaneously perform distributed addition and multiplication on blocks of ℓ packed secrets, thereby evaluating a function f on a batch of ℓ (independent) inputs. The new protocol requires $O(\ell)$ more players than the original one, to compensate for the higher degree of the polynomials used for sharing the secrets.

From here on, we will take ℓ to be some constant fraction of n, and apply the share packing technique to protocols that tolerate a constant fraction of corrupted players. In this setting of the parameters, the resulting protocol uses $\Omega(n)$ times less communication than ℓ repetitions of the original protocol, and still tolerates a (smaller) constant fraction of corrupted players.

The problem encountered by [21] when trying to go beyond multiple evaluations of the same function is that "splitting" a vector of packed secrets is an expensive operation. Thus, the above savings could not be directly translated to the case of evaluating a complex function on a single input. We explain below how one can get around this problem using the garbled circuit technique, once we have explained a second source of savings.

[7] The approach of Franklin and Yung was recently generalized, in the case of a passive adversary, to non-threshold adversary structures [38].

3.2 The "Easy PRG" Technique

All variants of our protocol make use of a pseudorandom generator (PRG). The main variant of the protocol further requires an "easy PRG", namely a PRG computable in NC^1 or similar complexity classes. (Here by PRG we refer to any nontrivial PRG that extends its seed by just a single bit.) Such a PRG is implied by most standard cryptographic intractability assumptions such as those related to factoring, discrete logarithms, and lattice problems (see [1]). Efficient constructions of PRGs with a low algebraic degree that expand their seed by a constant factor can make our protocol quite efficient in practice. Candidates for such PRGs can be found in [32, 3].

Our protocol will use the PRG in order to reduce the functionality f we wish to evaluate to a *low-degree* randomized functionality $\hat{f}$. (Here and in the following, the degree of a randomized functionality is its degree over GF(2), counting both inputs and random inputs towards the degree.) More precisely, we would like to obtain a low-degree randomized functionality $\hat{f}(x, r)$ with the following properties: (1) given the output of $\hat{f}(x, r)$ it is possible to efficiently *decode* $f(x)$; and (2) given the output $f(x)$ alone it is possible to efficiently *simulate* the output distribution of $\hat{f}(x, r)$ induced by a uniform choice of r, up to computational indistinguishability. The above properties of $\hat{f}$ allow to obtain a non-interactive *secure reduction* from f to $\hat{f}$: to evaluate f the players first invoke $\hat{f}$, and then apply the efficient decoder (from property (1) above) to recover the output of f.

The question of securely reducing general functionalities to low-degree functionalities was initially studied in [28] in an information-theoretic setting and was recently studied in a computational setting in [2]. The idea of [2] is to combine the "computational" circuit garbling technique of Yao [37] with "information-theoretic" randomization techniques of [28, 1]. The former reduces the evaluation of a circuit C to an evaluation of a degree-3 randomized functionality $\tilde{C}$ having an oracle access to a PRG; the latter then uses the easiness of the PRG for reducing $\tilde{C}$ to a degree-3 randomized functionality $\hat{C}$ which does not make use of a PRG oracle. We now describe the high-level details of these reductions that are useful for our purposes.

Computational reduction. Yao's technique (e.g., using its variants described in [35] or [2]) reduces the functionality $C(x)$ defined by a circuit C to a randomized functionality $\tilde{C}(x, \tilde{r})$ having the following structure. The random input $\tilde{r}$ includes $O(|C|)$ wire keys $s_{w,b} \in \{0,1\}^k$ (where w ranges over the wires and $b \in \{0,1\}$), as well as wire masks $\lambda_w \in \{0,1\}$. The functions computing the output bits of $\tilde{C}$ can be divided into three classes:

I. For each input bit of C, we have a function taking the input bit x_i and corresponding wire keys $s_{i,0}, s_{i,1}$ and outputting the key s_{i,x_i}. Overall, we have $|x|$ functions of type I, each with $O(k)$ input bits and $O(k)$ output bits, where the outputs can be viewed as degree-2 polynomials in the inputs.

II. For each NAND gate of C, with input wires α, β and output wire γ, we have a function mapping the 6 wire keys $s_{\alpha,b}, s_{\beta,b}, s_{\gamma,b}$ and the 3 wire masks

$\lambda_\alpha, \lambda_\beta, \lambda_\gamma$ to $O(k)$ outputs. Each of these outputs is a degree-3 polynomial acting on the $O(k)$ input bits mentioned above as well as the outputs of the PRG applied to the wire keys. (The PRG is applied here to expand the wire keys and use the result to encrypt other keys and wire masks.[8]) Overall, we have $|C|$ functions of type II, each with $O(k)$ input bits and $O(k)$ output bits, where the outputs can be viewed as degree-3 polynomials in the inputs given an oracle to a PRG.

III. For each output wire w of C, we have a function taking a wire mask λ_w and outputting the same value.

The crucial feature of the functionality $\tilde{C}$ is that it consists of $|C|$ parallel copies of *few* types of degree-3 functionalities, each having a *short* input and output. The inputs to these functionalities have to be replicated in a way that reflects the structure of the circuit C. Note, however, that $\tilde{C}$ makes use of a PRG oracle, which we would like to eliminate. This is done via the following information-theoretic reduction.

Information-theoretic reduction. Using information-theoretic randomization techniques of [28, 1], it is possible to reduce the randomized functionality $\tilde{C}(x, \tilde{r})$ defined above to a similar randomized functionality $\hat{C}(x, \hat{r})$ whose outputs are degree-3 polynomials in $x, \hat{r}$. This reduction relies on the underlying PRG being computationally easy. The functionality $\hat{C}$ has a similar high level structure to that of $\tilde{C}$ described above, except that the random input $\hat{r}$ of $\hat{C}$ is longer and includes $\mathrm{poly}(k)$ additional bits for every gate. Moreover, functions of type II now have $\mathrm{poly}(k)$ many inputs and outputs instead of k. Again, the inputs to all of the $O(|C|)$ functions are a suitable replication of the functionality's inputs $x, \hat{r}$, where the replication pattern depends on the structure of C.

We summarize the above with the following lemma:

Lemma 1. *[2] Suppose an "easy PRG" (e.g., a PRG in* NC^1*) exists. Then, the secure computation of an arbitrary circuit $C(x)$ can be reduced to the secure computation of a degree-3 randomized functionality $\hat{C}(x, \hat{r})$, where $\hat{C}$ outputs a concatenation of $|C|$ functions of types I, II, and III as above. This reduction is secure against an active, adaptive adversary corrupting any set of players. It is a* non-interactive *reduction, in the sense that computing C involves only a single invocation to $\hat{C}$ and local computation, without further communication.*

We are now ready to explain at a high level how the tools described above can be combined. Intuitively, Yao's garbled circuit technique transforms each gate in the circuit C to a table of encrypted values, such that with appropriately encrypted inputs, one can evaluate C on these inputs while learning only the intended result. The lemma above essentially says that the encrypted circuit can be created by evaluating (securely and in parallel) $|C|$ degree-3 functions, namely the functions of type II above. Moreover, these functions are all of the same form

[8] Here we assume that the PRG has a sufficiently high stretch. This assumption can be eliminated with an $O(k)$ multiplicative overhead to the total complexity; see [2].

because we may assume that all gates are NAND-gates, and the original garbling procedure treats every gate in the same way.

Now recall that evaluating many instances of the same function is exactly what Franklin and Yung's share packing technique allows us to do efficiently. There is one technical difficulty that needs to be overcome, however: the random inputs for the instances of the function have to be correlated (in fact, replicated) in a way that reflects the structure of C. Thus, in the plain malicious model (without preprocessing) we will need to devise a protocol for efficiently enforcing such complex correlations. This problem will be addressed in Section 5.

4 Scalable MPC in the Linear Preprocessing Model

In this section we show how the previous tools can be combined to obtain scalable MPC protocols in the linear preprocessing model. In fact, in this model we can get the *overall* complexity to be $\tilde{O}(|C|)$, assuming a constant number of output clients[9] and regardless of the number of input clients. For simplicity, from here on we will mostly address the *communication* complexity. A treatment of the computational complexity (along with relevant optimizations) appears in the full version. We also assume for simplicity that the functionality outputs the same value $C(x)$ to all output clients. Our main result in this setting is captured by the following theorem.

Theorem 1 (Scalable MPC in the linear preprocessing model). *Suppose an easy PRG exists. Then, in the linear preprocessing model, there is a general two-round MPC protocol tolerating $t = \Omega(n)$ malicious servers, in which the communication complexity for computing a circuit C involves $O(|C| \cdot \mathrm{poly}(k))$ bits sent to each output client, each input client must broadcast its (masked) inputs to the n servers, and there is no communication between the servers.*

Proof sketch: Let $\ell = \Theta(n)$ be the share packing parameter, as described in Section 3.1. The high level idea is to evaluate $\hat{C}(x,r)$ on a packed representation of its inputs and random inputs produced using ShareBlock. That is, we would like to evaluate ℓ copies of each type of output (I, II, or III) *in parallel* using the approach of [21]. To set up such an evaluation, we partition the (at most $|C|$) functions of each type into blocks of size ℓ. (For simplicity, assume wlog that the number of functions of each type is an exact multiple of ℓ, and so is the number of input bits held by each input client.) To evaluate ℓ instances of each function in parallel we replace each of the $\mathrm{poly}(k)$ input bits on which it depends by a block of ℓ bits, one from each instance. (That is, block i contains the i-th input bit from each instance.) We assume that the input bits x_i are all packed into the first $|x|/\ell$ blocks, referred to as input blocks, and that the o wire masks λ_w of

[9] This assumption can also be dispensed with by letting a single output client receive the (encrypted and authenticated) outputs of all clients, and then send to every other output client its share of the output. However, the resulting protocol only satisfies a relaxed notion of security, allowing the adversary to abort the computation.

the output wires are packed into the last o/ℓ blocks, referred to as output blocks. (We stress that in order to secret-share the bits belonging to a single block B, we form a block B' of ℓ elements in K where the i-th element equals the i-th bit in B, and secret-share B' using the procedure ShareBlock from Section 3.1; this should not be confused with packing several bits into one element of K.)

In addition to the input and output blocks, we have $O(|C| \cdot \text{poly}(k)/\ell)$ blocks containing bits of the random input $\hat{r}$, suitably replicated for allowing the parallel evaluation of the above 3 types of functions. A bit more precisely, this replicated assignment of random bits into blocks is formed as follows. First, the output bits of $\hat{C}$ are partitioned into groups of size ℓ, where the outputs in each group are computed by applying the same function g on some set of $k' = \text{poly}(k)$ bits of $\hat{r}$. (The function g is the same for each of the ℓ outputs in a group, but the sets of k' input bits may be different; note that since $\hat{C}$ employs only poly(k) distinct functions g, we can fit all outputs of $\hat{C}$ into $O(|C| \cdot \text{poly}(k)/\ell)$ groups.) The relevant bits of $\hat{r}$ are then placed into k' blocks so that the i-th output from the group is obtained by applying g to the i-th input bit from each block. Overall, each bit of $\hat{r}$ is placed into as many slots as the number of (type I, II, or III) functions in which it is involved. Note that the replication pattern of the bits of $\hat{r}$ depends on the circuit topology via the functions of type II.

Let $m = O(|C| \cdot \text{poly}(k)/\ell)$ denote the total number of blocks. To represent the replication requirements induced by the circuit structure it is convenient to view the $m\ell$ elements of the blocks as being colored according to their partition classes. That is, we have a coloring function $\phi_C : [m] \times [\ell] \to [\ell m]$ such that $\phi_C(i,j) = \phi_C(i',j')$ iff entry j of block i should be identical to entry j' of block i' (otherwise they are assumed to be independent). The entries of the input blocks will all be colored by unique colors (since each input is used only by a single function of type I), but all other entries (including those of the output wires) will be replicated.

We are now ready to describe the scalable protocol in the linear preprocessing model. We assume that the field K is of characteristic 2, and is of minimal size subject to $|K| > n$ (the requirement $|K| > n$ is imposed by ShareBlock). Let $\ell = \Theta(n)$ and $T = t + \ell - 1$. To enable non-interactive computation of degree-3 polynomials on blocks of ℓ packed secrets, it suffices to use $n > 2T$ players in the semi-honest case and $n > 5T$ players in the malicious case (as required for evaluating degree-3 polynomials with minimal interaction, cf. [16], Section 2.2). The protocol proceeds as follows.

Protocol 1 Scalable MPC protocol in linear preprocessing model:

1. Setup. *Using linear preprocessing, the players obtain packed shares of replicated random* binary *secrets, where the replication pattern is specified by ϕ_C. Moreover, each input client obtains the (random) secrets belonging to the blocks of inputs it owns. Finally, the players obtain packed shares of $O(m)$ independent* 0-blocks*, namely blocks whose ℓ secrets $(s_0, \ldots, s_{\ell-1})$ are all set to 0. These 0-blocks are secret-shared using polynomials of degree $3T$ and will be used in Round 3 to support secure multiplication.*

Note that each of the above distribution patterns can be expressed as a random codeword in a distributed linear code over GF(2)*, and thus can be implemented within the linear preprocessing framework. (The distributed codes used above are only linear over* GF(2) *and not over the extension field K, since the random secrets are restricted to take 0/1 values.)*

2. Round 1. *Each input client broadcasts to the servers a correction vector of length ℓ for each input block it owns, namely the exclusive-or of this block with a corresponding block of random secrets obtained during setup.*
3. Round 2. *First, each server locally corrects its share of each input block according to the corrections broadcasted in Round 1. For instance, if the correction for the first block is $c \in \{0,1\}^\ell$, server P_j finds the (unique) polynomial p of degree T such that $p(\mu_i) = c_i$ for $1 \leq i \leq \ell$ and $p(\omega_{j'}) = 0$ for $1 \leq j' \leq t$, and adds $p(\omega_j)$ to its share of the first block as received during setup. Following this stage, all inputs to $\hat{C}$ (including actual inputs x and random inputs $\hat{r}$) are shared between the servers.*

   Now the servers locally compute random shares of the outputs of $\hat{C}$ and send them to the output clients. This local computation is done by simply evaluating the corresponding degree-3 polynomials on the local (packed) shares and adding the (packed) shares of the 0*-blocks. (Note that the careful structuring of the blocks, defined by the replication pattern ϕ_C, ensures that the computation applied to each set of $k' = \mathrm{poly}(k)$ blocks is the same for each position i, $1 \leq i \leq \ell$, within the blocks.)*
4. Output. *Each output client reconstructs the outputs of $\hat{C}$ from the shares it received, using error-correction of Reed-Solomon codes to recover from possible errors caused by malicious servers. It then applies the efficient decoding procedure of $\hat{C}$ to recover the output of C.*

The security of Protocol 1 follows by combining (1) the security of the reduction from C to $\hat{C}$ (Lemma 1) and (2) the security of the above procedure for computing $\hat{C}$ (namely, the one defined by Protocol 1 without the final decoding step). The latter procedure is analogous to a similar protocol in [16], Section 2.2, except for the use of packed shares instead of standard shares. Thus, a formal proof of (2) can be easily obtained by generalizing the corresponding proof in [16]. Finally, the security of Protocol 1 is implied by (1) and (2) using a suitable composition theorem (e.g., from [10, 11]).

The communication complexity includes broadcasting the (masked) inputs by the clients, and n field elements communicated to each output client (one from each server) for each ℓ-tuple of output bits of $\hat{C}$. Overall, each output client receives $O(|\hat{C}| \cdot n/\ell) = O(|\hat{C}|) = O(|C| \cdot \mathrm{poly}(k))$ field elements, as required. □

We note that a variant of Protocol 1 (and also of our main protocol) can be based on an arbitrary PRG, using techniques of [5, 24] to directly evaluate the PRG-based functionality $\tilde{C}$ defined in Section 3.2 instead of the degree-3 functionality $\hat{C}$. This variant requires $\mathrm{poly}(k)$ rounds of interaction, and cannot go below the $\tilde{O}(n|C|)$ communication bound.

5 Scalable MPC in the Plain Model

In this section we implement Protocol 1 in the plain model. This amounts to securely emulating Step 1, namely the generation of replicated random secrets, without the trusted setup of linear preprocessing. In the semi-honest model, this can be done in a straightforward way by letting each player share random (packed) secrets replicated according to ϕ_C, and adding up the contributions of $t+1$ different players. Thus, for the semi-honest model we immediately get a scalable protocol with a total of $O(n|C| \cdot \mathrm{poly}(k))$ communication. In fact, it suffices to let only the *clients* participate in the generation of the random packed shares. (Indeed, even one uncorrupted client would suffice to keep the shared secrets hidden from the adversary.) So when the number of clients is constant, the communication is only $O(|C| \cdot \mathrm{poly}(k))$.

We turn to the question of securely generating the required linear correlations in the malicious model. We will begin by describing an efficient procedure for generating many *independent* instances of the *same* linear correlation. Later, in Section 5.1, we will extend this basic procedure to deal with the more challenging case of correlations induced by an arbitrary replication pattern ϕ_C, as required by the setup phase of Protocol 1.

The basic procedure we present below is similar to a subprotocol of Hirt and Maurer [24], and in particular uses their ideas of player elimination (also used, in different ways, in [25, 22, 26, 8]) and the idea of verifying a large batch of linearly-shared secrets by using randomized linear combinations. We will improve its efficiency by using distributed coin-flipping and a pseudorandom generator to succinctly generate the coefficients of a long (pseudo-)random linear combination. In the full version we describe an optimization of the asymptotic *computational* complexity of the protocol. (The corresponding subprotocol from [24] does not rely on distributed coin flipping nor a PRG; instead, it requires each player to separately communicate a long linear combination vector. Moreover, this subprotocol was implicitly analyzed in the non-adaptive case; an extension of its analysis to the adaptive case would involve an additional overhead.)

Generating independent instances of a linear correlation. Consider the randomized functionality $\mathsf{RandDLC}(L)$, which distributes to the players shares of a random codeword from a distributed linear code $L = (V, \pi_1, \ldots, \pi_n)$. We want to securely compute m independent instances of $\mathsf{RandDLC}(L)$ at a low cost.

To get the desired efficiency with a minimal amount of interaction, we will not insist on securely computing the desired functionality in the strict sense, but rather settle for the following relaxation. At the end of the protocol, we can tolerate a small set B of (say, at most $2t$) "eliminated" players, such that only the outputs of honest players *outside* B should be consistent with the functionality. Moreover, this set B is publicly known. (Jumping ahead, we will require that the same global set B apply jointly to all of the subprotocols we use.) This notion of security can be formally captured by relaxing the standard simulation requirement to consider, in addition to the view of corrupted players, the joint outputs of all honest players outside the set B (instead of *all* honest players in the standard simulation-based definitions). For instance, consider the case of applying $\mathsf{RandDLC}$ to secret-share a block of random secrets. In this case, such a definition would imply that the outputs of all honest players *outside* B are consistent with a valid sharing of the same block, and moreover that the adversary learns nothing about the secrets in the block.

This relaxed notion of security is implicit in previous protocols (e.g., [24, 22]). It does not compromise the security of our high level protocols, since: (1) all relevant functionalities are *robust*, in the sense that a "correct" output can be efficiently decoded from a partial output; and (2) such a decoding procedure will be employed by the output clients in the high level protocol. Thus, from the point of view of the higher level application, the missing shares of B do not make any difference.

We turn to describe our amortized protocol for $\mathsf{RandDLC}(L)$. In fact, we start with an even simpler *deterministic* functionality we denote by $\mathsf{DLC}^P(L)$.

Definition 1 (Functionality $\mathsf{DLC}^P(L)$). *Let $L = (V, \pi_1, \ldots, \pi_n)$ be a distributed linear code, where V is spanned by the columns of a matrix M. Let P be a player (an input client or a server) referred to as a* dealer. *The functionality $\mathsf{DLC}^P(L)$ is defined as follows:*

- **Inputs:** *P holds an input y, defining a vector $v = My \in V$. Other players have no input.*
- **Outputs:** *Each server P_j outputs $v(j)$.*

The secure computation of $\mathsf{DLC}^P(L)$ is trivial in the semi-honest model, since P can simply send to each server j its share $v(j)$. It is nontrivial in the malicious model, since we need to ensure that the outputs of honest servers are consistent with L, and that they are furthermore consistent with v when P is uncorrupted. Our motivation for securely computing $\mathsf{DLC}^P(L)$ is that this will later be used as a building block for computing $\mathsf{RandDLC}(L)$. Specifically, $\mathsf{RandDLC}(L)$ can be reduced to multiple instances of $\mathsf{DLC}^P(L)$ by adding together random vectors contributed by different dealers P. Thus, from here on we can focus on the goal of amortizing the cost of securely computing m instances of $\mathsf{DLC}^P(L)$, on input vectors $v_1, \ldots, v_m$. This is done as follows.

Protocol 2 Efficient amortized implementation of $\mathsf{DLC}^P(L)$:

- **Inputs:** *P holds m inputs $y_1, \ldots, y_m$, defining m vectors $v_i = My_i \in V$. Other players have no inputs.*
- **Outputs:** *Each server P_j outputs $v_1(j), \ldots, v_m(j)$.*

1. Share distribution. *P sends to each server P_j, $1 \leq j \leq n$, its share $v_i(j)$, $1 \leq i \leq m$, of each vector v_i. In addition, it similarly distributes σ* random *vectors $r^1, \ldots, r^\sigma \in_R L$ (which will be used as "blinding" vectors to protect the privacy of $v_1, \ldots, v_m$). To resist an* adaptive *adversary, we will need to let $\sigma = n + k$, where k is the security parameter.*

2. Coin-flipping. *The servers use a constant-round multiparty coin-flipping protocol (e.g., a 3-round protocol based on the 2-round VSS protocol from [22]) to pick a σ-tuple of random k-bit seeds $(s^1, \ldots, s^\sigma)$. Each seed s^h is locally expanded by each server, via a PRG, to a vector R^h in $\{0,1\}^m$, specifying a (pseudo-)random linear combination of the v_i's.*
3. Proving. *P computes and broadcasts the σ linear combinations specified by the vectors R^h, masking each with a different blinding vector r^h. That is, it computes and broadcasts the σ codewords $u^h = r^h + \sum_{i=1}^m R_i^h v_i$. (The vectors u^h serve as a succinct distributed witness for the validity of the vectors v_i.) Each P_j verifies that all u^h are valid codewords; otherwise P is disqualified and all honest servers output* 0.
4. Complaining. *Each P_j verifies that every u^h is consistent with the linear combination R^h and the $m + \sigma$ shares it received. If some inconsistency is found, P_j broadcasts a complaint.*
5. Output. *Let B be the set of servers who broadcasted a complaint. If $|B| > t$, the dealer P is disqualified and all honest servers output 0. Otherwise, each P_j outputs its shares of $v_1, \ldots, v_m$ and discards its shares of $r^1, \ldots, r^\sigma$ (which were used for blinding).*

Lemma 2. *Protocol 2 satisfies the following:*

1. *If P is honest then it is never disqualified, the outputs of the honest servers are consistent with the v_i's, and the set B includes only corrupted servers.*
2. *Suppose P is corrupted. If P is not disqualified, then (except with negligible probability) all outputs of honest servers outside B are consistent with L. In other words, the event that P is not disqualified and the outputs of honest servers outside B are inconsistent occurs with negligible probability in k.*
3. *If P is honest, the adversary learns nothing about the vectors v_i except the shares of corrupted servers. (Moreover, its view can be efficiently simulated given these shares.)*

Proof sketch: Fact (1) follows by inspection of the protocol, and (3) follows from the use of the blinding vectors r^h. It remains to show that (2) holds. For this, we bound the probability that there is *some* set H of honest servers whose shares are inconsistent yet pass all tests. Suppose first that the linear combinations R^h were entirely random, and fix a possible set $H \subseteq [n]$ of honest servers. If some share vector received by H, wlog $v_1(H)$, is inconsistent with L, then a random linear combination R^h of all vectors is inconsistent with L with probability at least 1/2. (Indeed, conditioned on the linear combination v' of the remaining vectors, there are two cases: (a) if $v'(H)$ is consistent with L then $[1 \cdot v_1 + v'](H)$ is inconsistent with L; (b) if $v'(H)$ is inconsistent with L then $[0 \cdot v_1 + v'](H)$ is inconsistent with L.) It follows that H will pass all σ consistency tests with at most $2^{-\sigma}$ probability. Since $\sigma = n + k$, it follows by a union bound that there is no set H of honest servers who receive inconsistent shares and do not complain, except with negligible probability. Specifically, this failure probability is bounded by $2^n \cdot 2^\sigma = 2^{-k}$. Finally, by the security of the PRG the above analysis remains the essentially the same when each R is pseudorandom (e.g., replace 1/2 in the above analysis by 0.4). □

Note that when m is large, the communication complexity of the protocol is dominated by the length of the inputs and is thus essentially optimal. Also note that one could instantiate the PRG with an (unconditionally secure) ϵ-biased generator [33].

The multiple-dealer case. To implement $\mathsf{RandDLC}(L)$ we need to use multiple (parallel) invocations of the above $\mathsf{DLC}^{P_i}(L)$ protocol with different dealers P_i. A major concern in this case is that each invocation may result in a different set B_i of eliminated players. Indeed, the adversary can arrange that for each corrupted P_i, the set B_i will include a disjoint set of t honest players. Our goal is to obtain a publicly known set B of at most $2t$ players, whose elimination would result in all honest players holding consistent shares from all remaining invocations of DLC. Moreover, to achieve this goal we can afford to disqualify up to t honest dealers, overriding their inputs with 0. This further relaxation will not affect the security of our applications, in which DLC will be invoked with random and independent codewords rather than ones that depend on the inputs. Specifically, when using n invocations of DLC for implementing $\mathsf{RandDLC}$, each player will distribute a random codeword and output the sum of the received shares. In this case, eliminating a few honest players will have no effect on security. (In fact, it would suffice that $2t+1$ players act as dealers.)

The above is easy to achieve using the following player elimination approach. Let B_i denote the union of all sets B induced by DLC invocations in which P_i acts as the dealer. It follows from Lemma 2 that either P_i is corrupted or all players in B_i are corrupted. Thus, we can eliminate both P_i and some $P_j \in B_i$ and include them in B. As a result, the size of B grows by 2 and the number of remaining corrupted players is reduced by at least 1. After no more than t such elimination steps all public inconsistencies are resolved and B contains at most $2t$ players. At the end of this procedure, all dealers belonging to the (publicly known) B are eliminated. We are guaranteed that the outputs of all non-eliminated honest players are consistent in all DLC invocations in which one of them serves as a dealer.

In the setting where the number of clients is constant, we can avoid the above procedure by letting only the clients act as dealers. (This suffices because there is no need to protect against an adversary that corrupts all clients.) In this case we can afford to let the final B be the union of all B_i associated with clients that were not disqualified, while still tolerating a constant fraction of corrupted servers ($t = \Omega(n)$). Since this approach does not involve an elimination of honest clients, it can also be used to verify shares of the *inputs*, thus avoiding the need to broadcast all inputs.

5.1 Generating Packed Replicated Secrets

The above $\mathsf{RandDLC}$ protocol suffices for generating the packed shares of 0-blocks required by the setup phase of Protocol 1. Indeed, to generate these blocks, let

L be the space of degree-T polynomials over K which evaluate to 0 on all points μ_i, $1 \leq i \leq \ell$, where each P_j is assigned the value of the polynomial at point ω_j. The challenge is to efficiently generate m blocks of random binary secrets *subject to the replication pattern defined by* ϕ_C. Our main observation is that testing if a shared m-tuple of blocks is consistent with the "complex" replication pattern ϕ_C can be reduced, via local computation (and no communication), to testing $O(\ell^2)$ instances of a "repetitive" replication pattern. The latter can be efficiently handled using the testing procedure in Protocol 2. The extra cost of $O(\ell^2)$ elementary tests will be amortized over a large number of blocks $m \gg \ell^2$, and thus will not effect the amortized communication. We stress that the above holds for an arbitrary replication pattern ϕ_C, and thus does not depend on the circuit topology.

The idea for this reduction is the following. Consider the case where a single dealer P distributes m blocks of ℓ secrets. We want to verify that these are indeed *binary* secrets that are *consistent with* ϕ_C. The replication condition ϕ_C can be expressed as the conjunction of at most $m\ell$ atomic conditions of the form "$\phi_C(i,j) = \phi_C(i',j')$". We refer to such a condition as being of type (j,j'), indicating that the j-th secret in one block is compared with the j'-th secret in another, not necessarily distinct, block. There are at most ℓ^2 different types of atomic conditions, where each block is involved in at most $O(\ell)$ types (assuming C has constant fan-out).

The key idea is that each condition of type (j,j') can be tested by letting the servers locally concatenate their two shares from the two relevant blocks (i,i'), and then (jointly) test that the combined shares belong to some linear space $L_{j,j'}$. More specifically, let v, v' represent share vectors corresponding to blocks i, i' respectively, and let (v, v') be the distributed vector obtained by letting each server locally concatenate its shares of v, v' in that order. Then, the set of vectors (v, v') satisfying the above atomic replication constraint can be expressed as a test of membership in a distributed linear code $L_{j,j'}$ over GF(2). Indeed, if both (u, u') and (v, v') satisfy the atomic replication constraint, then so does $(u + v, u' + v')$. Finally, we can batch together all atomic conditions of the same type and test them together using the testing procedure defined by Steps 2-5 of Protocol 2. This gives a procedure for verifying the validity of the m blocks distributed by a dealer P_i via ℓ^2 (parallel) invocations of the previous testing procedure. This procedure will either disqualify the dealer, or output a set B_i of at most t servers such that all shares of honest servers outside B_i are consistent packed shares of replicated binary secrets satisfying ϕ_C.

More concretely, for each player P, acting as a dealer, we run in parallel the following procedure.

Protocol 3 Generating packed random secrets consistent with replication pattern ϕ_C:

1. *P generates m blocks of random binary secrets subject to the replication constraints ϕ_C, picks a random degree-T polynomial encoding each block, and sends to each P_j the value of each of the m polynomials at point ω_j. In*

addition, a random "blinding" vector $r^h \in_R L_{j,j'}$ should be distributed for each type (j, j').

2. *Run Steps 2-4 of Protocol 2, testing that all shared blocks encode ℓ-tuples of binary secrets.*
3. *The servers write ϕ_C as a conjunction of (at most $m\ell$) atomic equality conditions, represented by a set $A \subseteq ([m] \times [\ell])^2$. For each type of atomic condition $(j, j') \in [\ell]^2$, do the following (in parallel):*
   For every pair (i, i') such that $((i, j), (i', j')) \in A$, let each server concatenate its share of block i and its share of block i'. Run Steps 2-4 of Protocol 2 to test that all concatenated vectors (each of length $2n$) are in the linear space $L_{j,j'}$ corresponding to atomic conditions of type (j, j').
4. *Run Step 5 of Protocol 2 to produce the outputs.*

Protocol 3 may result in eliminating the dealer P, or producing a set B of at most t servers. The final protocol will use parallel invocations of the above protocol with each player acting as a dealer, in an analogous way to the amortized RandDLC protocol described above. As in RandDLC, these invocations produce a global set B of eliminated players. The final m-tuple of blocks is obtained by summing up the contributions of all non-eliminated dealers. Again, in the case of a constant number of clients, we let the clients serve as dealers and refrain from eliminating clients that were not disqualified as (certain) cheaters.

The combined cost of all the testing procedures is bounded by a fixed polynomial in n, independently of m. As a result, the amortized complexity of the share generation protocol is the same as in the semi-honest case. Thus, we have:

Theorem 2 (Scalable protocol in plain model). *Suppose an easy PRG exists. Then, in the plain model, there is a general constant-round MPC protocol tolerating $t = \Omega(n)$ malicious servers, in which the communication complexity for computing a circuit C involves $O(|C| \cdot n \cdot \text{poly}(k))$ bits of communication, and each input client must broadcast its (masked) inputs to the n servers. If the number of clients is constant, the complexity is reduced to $O(|C| \cdot \text{poly}(k))$.*

Proof sketch: The main protocol is obtained by combining Protocol 1 with the above share generation protocols, namely:

1. Use parallel invocations of Protocol 3 and server elimination (as described above) to generate shared blocks of random secrets, replicated according to ϕ_C, substituting the linear preprocessing of Step 1 in Protocol 1. The outputs of the servers are consistent with the required pattern except, perhaps, for a set B of $O(t)$ eliminated servers.
2. Run Steps 2-4 of Protocol 1 using the outputs from the previous step to substitute linear preprocessing.

A proof of security can proceed as follows. If Step 1 were a fully secure protocol for the corresponding randomized functionality (i.e., with $B = \emptyset$), then the security of the protocol would follow directly from that of Protocol 1 and a composition theorem of [10]. The $O(t)$ outputs of servers in B (resulting from

Step 1) have no effect on the outputs of output clients in the end of Step 2 because of the robustness property of this step. Moreover, the effect B has on the adversary's view can be easily incorporated into the simulation. $\square$

References

[1] B. Applebaum, Y. Ishai, and E. Kushilevitz. Cryptography in NC^0. In *Proc. FOCS 2004*, pages 165-175.

[2] B. Applebaum, Y. Ishai, and E. Kushilevitz. Computationally private randomizing polynomials and their applications. In *Proc. CCC 2005*, pages 260-274.

[3] B. Applebaum, Y. Ishai, and E. Kushilevitz. On Pseudorandom Generators with Linear Stretch in NC^0. In *Proc. RANDOM 2006*.

[4] O. Barkol and Y. Ishai. Secure Computation of Constant-Depth Circuits with Applications to Database Search Problems. In *Proc. Crypto 2005*, pages 395-411.

[5] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *Proc. Crypto 1991*, pages 420-432.

[6] D. Beaver, J. Feigenbaum, J. Kilian, and P. Rogaway. Security with low communication overhead. In *Proc. Crypto 1990*, pages 62-76.

[7] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *Proc. STOC 1990*, pages 503-513.

[8] Z. Beerliova and M. Hirt. Efficient Multi-Party Computation with Dispute Control. In *Proc. TCC 2006*, pages 305-328.

[9] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. STOC 1988*, pages 1-10.

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

[11] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proc. FOCS 2001*, pages 136-145.

[12] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. STOC 1988*, pages 11-19.

[13] Richard Cleve. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *Proc. STOC 1986*, pages 364-369.

[14] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proc. TCC 2005*, pages 342-362.

[15] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proc. Eurocrypt 2001*, pages 280-299.

[16] I. Damgård and Y. Ishai. Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator. In *Proc. Crypto 2005*, pages 378-394.

[17] I. Damgård, and J. Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Proc. Crypto 2003*, pages 247-264.

[18] S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM,* 28(6):637–647, 1985.

[19] P. Feldman and S. Micali. An Optimal Algorithm for Synchronous Byzantine Agreement. *SIAM. J. Computing*, 26(2):873–933, 1997.

[20] M. K. Franklin and S. Haber. Joint Encryption and Message-Efficient Secure Computation. In *Proc. Crypto 1993*, pages 266-277. Full version in *Journal of Cyptoglogy* 9(4): 217-232 (1996).

[21] M. K. Franklin and M. Yung. Communication Complexity of Secure Computation. In *Proc. STOC 1992*, pages 699-710.

[22] R. Gennaro, Y. Ishai, E. Kushilevitz and T. Rabin. The Round Complexity of Verifiable Secret Sharing and Secure Multicast. In *Proc. STOC 2001*, pages 580-589.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game (extended abstract). In *Proc. STOC 1987*, pages 218-229.

[24] M. Hirt and U. M. Maurer. Robustness for Free in Unconditional Multi-party Computation. In *Proc. Crypto 2001*, pages 101-118.

[25] M. Hirt, U. M. Maurer, and B. Przydatek. Efficient Secure Multi-party Computation. In *Proc. Asiacrypt 2000*, pages 143-161.

[26] M. Hirt and J. B. Nielsen. Upper Bounds on the Communication Complexity of Optimally Resilient Cryptographic Multiparty Computation. In *Proc. Asiacrypt 2005*, pages 79-99.

[27] M. Hirt and J. B. Nielsen. Robust Multiparty Computation with Linear Communication Complexity. These proceedings.

[28] Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *Proc. FOCS 2000*, pages 294-304.

[29] M. Jakobsson and A. Juels. Mix and Match: Secure Function Evaluation via Ciphertexts. *Proc. Asiacrypt 2000*, pages 162-177.

[30] J. Katz and C.-Y. Koo. On Expected Constant-Round Protocols for Byzantine Agreement. These proceedings.

[31] Y. Lindell, A. Lysyanskaya, and T. Rabin. Sequential composition of protocols without simultaneous termination. In *Proc. PODC 2002*, pages 203-212.

[32] E. Mossel, A. Shpilka, and L. Trevisan. On ϵ-biased generators in NC^0. In *Proc. FOCS 2003*, pages 136-145.

[33] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993. Preliminary version in Proc. STOC '90.

[34] M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *Proc. STOC 2001*, pages 590-599.

[35] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proc. 1st ACM Conference on Electronic Commerce*, pages 129-139, 1999.

[36] A. Shamir. How to share a secret. *Commun. ACM*, 22(6):612–613, June 1979.

[37] A. C. Yao. How to generate and exchange secrets. In *Proc. FOCS 1986*, pages 162-167.

[38] Z. Zhang, M. Liu, and L. Xiao. Parallel Multi-Party Computation from Linear Multi-Secret Sharing Schemes. In *Proc. Asiacrypt 2005*.