Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator[*]

Ivan Damgård[†] Yuval Ishai[‡]

August 10, 2005

Abstract

We present a constant-round protocol for general secure multiparty computation which makes a *black-box* use of a pseudorandom generator. In particular, the protocol does not require expensive zero-knowledge proofs and its communication complexity does not depend on the computational complexity of the underlying cryptographic primitive. Our protocol withstands an active, adaptive adversary corrupting a minority of the parties. Previous constant-round protocols of this type were only known in the semi-honest model or for restricted classes of functionlities.

1 Introduction

General secure computation is often perceived as being inherently impractical. One valid reason for this perception is the fact that all current protocols either require many rounds of interaction (e.g., [20, 5, 27, 35, 13, 22]), or alternatively require only a constant number of rounds but make use of expensive zero-knowledge proofs for each gate of the circuit being computed (e.g., [40, 4, 28, 6, 26, 34, 25]). Indeed, in all constant-round protocols from the literature, players need to provide zero-knowledge proofs for statements that involve the computation of a pseudorandom generator or other cryptographic primitives on which the "semi-honest" version of the protocol relies. Thus, these protocols make a *non-black-box* use of their underlying cryptographic primitives. We stress that this holds for all settings of secure computation with security against malicious parties, both in the two-party and in the multi-party case. The only exceptions to this general state of affairs are unconditionally secure protocols that apply to restricted classes of functionalities such as NC^1 [2] or protocols that require an exponential amount of computation [3].

In this work we consider the setting of multiparty computation with an honest majority, and present a general constant-round protocol that makes a *black-box* use of a pseudorandom generator.[1] Similarly to all general constant-round protocols from the literature, our protocol relies on Yao's

[*]An abridged version of this paper appears in the proceedings of Crypto 2005.

[†]Computer Science Department, Aarhus University, `ivan@daimi.au.dk`. 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.

[‡]Computer Science Department, Technion, `yuvali@cs.technion.ac.il`. Research supported by Israel Science Foundation grant 36/03.

[1]Here we assume the standard model of secure point-to-point channels. In the setting of open channels, our protocol can be adapted to make a black-box use of (a suitable form of) encryption.

garbled circuit technique [40], which was later adapted to the multi-party setting by Beaver, Micali, and Rogaway [4]. The latter "BMR protocol" requires players to verifiably secret-share seeds to a PRG as well as the outputs of the PRG on these seeds. To ensure that this is done correctly, the protocol makes a non-black-box use of the PRG by requiring players to prove via (distributed) zero-knowledge that the shared seeds are consistent with the shared PRG outputs. We get around this problem by modifying the basic structure of the BMR protocol, using "distributed symmetric encryption" and error-correction to replace the zero-knowledge proofs. Before providing a more detailed account of our results we give some more background to put them in context.

Black-box reductions in cryptography. Most reductions between cryptographic primitives are (fully) black box, in the sense that they implement a primitive A by using some other primitive B as an oracle, without depending on the implementation details of B. Moreover, the security proof of such reductions is also black-box in the sense that an adversary breaking A can be used as an oracle in order to break the underlying primitive B. (See [36] for a more detailed definition and discussion.) In contrast, a non-black-box reduction can use the "code" of B when implementing A. Most examples for non-black-box reductions in cryptography are ones in which the construction of A requires parties to prove in zero-knowledge statements that involve the computation of the underlying primitive B. For instance, the construction of an identification scheme from a one-way function [14] makes a non-black-box use of the one-way function. A rich line of work, originating from [24], uses oracle separations to rule out the existence of (various forms of) black-box reductions. Most notably, it is shown in [24] that there is no black-box reduction from key agreement to a one-way function. A common interpretation of such results is that they rule out the existence of "practically feasible" reductions. Indeed, all known examples for non-black-box reductions in cryptography involve a considerable overhead. In the context of cryptographic protocols, this overhead typically involves not only local computation but also communication: the *communication* complexity of the protocol B depends on the *computational* complexity of the underlying primitive A.[2] Our work provides further demonstration for the usefulness of distinguishing between the two types of reductions.

Constant-round secure computation. The question of implementing secure computation in a constant number of rounds has attracted a considerable amount of attention. The first general constant-round protocol for secure two-party computation was given by Yao [40]. Yao's original protocol considered only the case of semi-honest parties; an extension to the case of malicious parties (equivalently, an *active* adversary) was given by Lindell [28]. While Yao's original protocol makes a black-box use of the underlying primitives (a pseudorandom generator and oblivious transfer), the protocol from [28] relies on the methodology of Goldreich, Micali, ad Wigderson [20] and thus makes a non-black-box use of these primitives. Recently, Katz and Ostrovsky obtained a two-party protocol with an optimal exact round complexity [25].

An extension of Yao's protocol to the case of multiparty computation with an honest majority was given by Beaver, Micali, and Rogaway [4] (see also [37, 39]). Similarly to the two-party case, the BMR protocol makes a black-box use of a PRG in the semi-honest case and a non-black-box use of a PRG in the malicious case. (Because of the honest majority assumption, the protocol does not need to rely on OT.) For the case of a *dishonest majority*, constant-round multiparty protocols were recently obtained by Katz et al. [26] and by Pass [33]. (The latter protocol also achieves bounded

[2]In some cases it is possible to reduce the communication overhead by using communication-efficient zero-knowledge *arguments* (cf. [31]). However, this approach would make the computational overhead even higher.

concurrent security, extending a previous two-party protocol of Pass and Rosen [34].) The above protocols for the case of dishonest majority allow the adversary to prevent honest parties from receiving their output, even when it only corrupts a minority of the parties. Like the two-party protocols, they follow the GMW methodology and thus make a non-black-box use of the underlying primitives.

Finally, there has also been a considerable amount of work on *unconditionally* secure constant-round multiparty computation in the case of an honest majority (e.g. [2, 15, 23, 9]). Unfortunately, all known protocols in this setting can only be efficiently applied to restricted classes of functionalities such as NC^1 or non-deterministic logspace.

1.1 Our Results

We consider the model of computationally secure multiparty computation against an active, adaptive adversary corrupting up to $t < n/2$ players. Our default network model assumes secure point-to-point channels and the availability of broadcast (see more on that later). As stated above, our main result is a new "black-box feasibility" result. Specifically, we construct the first general constant-round protocol which makes a black-box use of a PRG (equivalently, using [21], a black-box use of a one-way function). Since much of our motivation comes from the goal of making secure computation more efficient, we also attempt to minimize the amount of interaction and communication required by our protocols. To this end, it is convenient to cast the protocols in the following "client-server" framework.

The client-server model. We divide the players into "input clients" who provide inputs, "output clients" who receive outputs, and "servers" who perform the actual computation. The security of the protocol should hold as long as at most t servers are corrupted, regardless of the number of corrupted clients. These three sets of players need not be disjoint, hence this is a strict generalization of the standard MPC framework in which all parties play all three roles. It also represents a likely scenario for applying MPC in practice, using specialized (but untrusted) servers to perform the bulk of the work. We stress again that this is just a refinement of the standard model. The main advantage of this refinement, besides conceptual clarity, is that it allows to decouple the number of "consumers" from the required "level of security". (The latter depends on the number of servers and the security threshold.) For instance, we can have just two clients and many servers (which may be viewed a distributed implementation of two-party computation), or a very large number of clients and only few servers. The latter might be the most realistic setting for secure computations involving inputs from many players.

Linear preprocessing. We present our main protocol in two stages. First, we present a protocol in what we call the "linear preprocessing model". In this model, it is assumed that there is a trusted setup phase where a dealer can provide clients and servers with linearly-correlated resources, e.g., Shamir-shares of random secrets. Then we use standard subprotocols for emulating the trusted setup in the plain model. The linear preprocessing model is motivated by the pseudorandom secret-sharing technique of [11]: when the number of servers is small, linear preprocessing can be emulated using a "once and for all" setup phase in which (roughly $\binom{n}{t}$) replicated and independent seeds are given to the players. Following this setup, the players can locally generate the required correlated shares without further interaction.[3]

[3]The method of [11] was only proved to be secure in the case of a non-adaptive adversary. Thus, when relying on pseudorandom secret sharing our protocol loses its provable adaptive security.

Our protocols. Our main protocol in the linear preprocessing model requires only two communication rounds when $t < n/5$. In the first round each input client broadcasts its masked inputs to the servers, and in the second round the servers send to each output client a total of $O(n^2|C|k)$ bits, where $|C|$ is the size of the circuit being computed and k is a security parameter.[4] In the plain model, one can obtain similar protocols at the cost of a higher communication complexity and additional rounds of interaction. When $t < n/5$, it suffices to use 3 rounds of interaction by relying on a VSS protocol from [17]. Alternatively, it is possible to tolerate $t < n/3$ or even $t < n/2$ malicious servers at the expense of further increasing the communication and the (constant) number of rounds. When $t < n/3$, a communication complexity of $O(n^3|C|k)$ bits (including the cost of broadcasts) can be obtained by using the techniques of [22].

In the case of computing a randomized functionality which has no inputs, the 2-round protocol in the linear preprocessing model becomes totally non-interactive when combined with pseudorandom secret-sharing. That is, to securely compute such a functionality it suffices for each server to send a single message to each output client, without using any broadcasts. Such non-interactive protocols can be used to obtain efficient distributed implementations of a trusted dealer in a wide range of applications.

On the use of broadcast. As in most of the MPC literature, our network model assumes the availability of broadcast as an atomic primitive. However, using the (expected) constant-round broadcast protocol of Feldman and Micali [16, 29], our protocols can be turned into (expected) constant-round protocols also in the point to point model. Concerning the communication complexity, since it is possible to implement our protocol so that the number of broadcasts involved is independent of $|C|$ (using the techniques of [22]), one can get the same (amortized) communication complexity in the point-to-point model. Moreover, in the typical scenario where the number of servers is small, even a "brute-force" simulation of the broadcasts will not have a major impact on efficiency.

Organization. The remainder of the paper is organized as follows. In Section 2 we define our security model and preprocessing models, and present some standard subprotocols in these models. Our main protocol and its variations are presented in Section 3, where the underlying distributed encryption idea is highlighted in Section 3.3. Finally, Appendix A contains a discussion of the case $t < n/2$ and Appendix B addresses the case of general preprocessing. Due to our elaborate use of techniques from previous works (mostly in the context of *information-theoretic* multiparty computation), we omit some of the low-level details and assume the reader's familiarity with standard MPC techniques from the literature.

2 Preliminaries

The Model. We consider a system consisting of several players, who interact in synchronous rounds via authenticated secure point-to-point channels and a broadcast medium. Players can be designated three different roles: *input clients* who hold inputs, *output clients* who receive outputs, and

[4]If there are many output clients with (potentially different) outputs it is possible to reduce the overhead of sending $O(n^2|C|k)$ bits to each output client by using a standard reduction to a single-output functionality. That is, the servers reveal to the first output client authenticated versions of all outputs, each masked with a key that is revealed only to the corresponding output client. Then the first output client sends to the remaining ones their authenticated masked outputs. In contrast to our main protocols, this variant only satisfies a relaxed security requirement, allowing an adversary corrupting the first output client to abort.

servers who may be involved in the actual computation. As discussed in Section 1.1, this is just a generalization of the standard model, where each party can play all three roles. We denote the number of servers by n. 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.[5] (In contrast, we will employ sub-protocols 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. (There is no restriction on the number of corrupted clients.) We refer the reader to, e.g., [7] for the standard definition of security in this model.

Our protocols will employ secret-sharing over a finite field $K = \mathrm{GF}(2^k)$, where k is a security parameter that will be used as the length of a seed to a PRG. Slightly abusing notation, each server P_i is assigned a unique nonzero value $i \in K$.

2.1 Linear Preprocessing

As discussed in Section 1.1, it will be convenient to describe and analyze our protocols in the *linear preprocessing model,* where we allow some restricted trusted setup as described below. The protocols can then be converted to the *plain model*, where no setup assumptions or preprocessing are allowed, at the price of some efficiency loss.

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 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 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 such a dealer would exist in practice. This is only a convenient abstraction, that can be formalized as an ideal functionality. We later separately look at how such a dealer may be implemented.

Note that in Shamir's secret sharing scheme, shares are computed as linear functions of the secret and random elements chosen by the dealer. We may therefore assume that the dealer can give to players Shamir shares of a random secret or of 0. More concretely, we assume that the following subroutines are available. When they are called in our protocol descriptions that follow, this should be taken to mean that the players retrieve from their preprocessed material values as specified below.

$\mathsf{RandSS}(t)$ Each server P_i obtains $f(i)$, where f is a random polynomial over K of degree at most t.

$\mathsf{RandSS}_0(t)$ Same as $\mathsf{RandSS}(t)$, except that f is subject to the restriction that $f(0) = 0$.

$\mathsf{RandSS}_{bin}(t)$ Same as $\mathsf{RandSS}(t)$, except that f is subject to the restriction that $f(0)$ is either 0 or 1. Note that this correlation pattern is linear over GF(2).

$\mathsf{RandSS}^P(t)$ Same as $\mathsf{RandSS}(t)$, except that player P additionally receives the polynomial f.

[5]Standard reductions from the general case to this special case involve interaction between output clients. This can be avoided by directly generalizing the protocol to the randomized multi-output case.

$\mathsf{RandSS}^P_{bin}(t)$ Same as $\mathsf{RandSS}_{bin}(t)$, except that P additionally receives f.

If the number of servers is small, all this can implemented efficiently using the share conversion technique from [19, 11]. This requires a "once and for all" setup, where a set of random seeds are generated, and each player receives a subset of these seeds. Then using any pseudorandom function, the players can execute any number of calls to the above subroutines without having to communicate. Each such call requires applying a local computation on the pre-distributed seeds. The price to pay for this is that the local storage and computation grow linearly with $\binom{n}{t}$. However, this overhead can be tolerated when the number of servers is small. In any case, the approach is only known to be secure against static adversary. We note that the setup phase of distributing replicated seeds can be implemented very efficiently (e.g., using a secure multicast protocol from [17]).

As discussed in Section 1.1, the linear preprocessing model is motivated by the pseudorandom secret-sharing technique from [11] (see also [19]). When the number of servers is small, a "once and for all" setup is sufficient for enabling players to execute any number of calls to the above subroutines without having to communicate.

One can emulate the linear preprocessing model in the plain model using constant-round interaction between players. This is trivial for a passive adversary, and can be done for an active adversary based on standard verifiable secret sharing schemes from the literature (e.g., [5, 12, 10]). In particular, [12] shows how to build VSS from any linear secret sharing scheme, and this can conveniently be used to implement $\mathsf{RandSS}_{bin}(t)$ using VSS over GF(2).

Our protocols will invoke the following variants of VSS as subroutines.

$\mathsf{VSS}^P(t)$ Player P has a value $s \in K$ as private input. The goal is for each server to receive a Shamir share of s. Using linear preprocessing, this only requires a single round of broadcast: in the setup phase we invoke $\mathsf{RandSS}^P(t)$. Then the value $r = f(0)$ can be computed by P, and P broadcasts $z = s - r$. Each server P_i takes $z + f(i)$ to be his private output.

$\mathsf{VSS}^P_{bin}(t)$ Player P has private input a value $b \in \mathrm{GF}(2)$. The goal is for each server to receive a Shamir share of b (computed over K). This can be implemented similarly to $\mathsf{VSS}^P(t)$, replacing $\mathsf{RandSS}^P(t)$ by $\mathsf{RandSS}^P_{bin}(t)$.

2.2 Secure Computation of Low-Degree Polynomials

In the linear preprocessing model, we now show how to securely compute the following functionality, which will be useful later.

The functionality is defined by $Q()$, a degree d polynomial over K in l variables $x_1, ..., x_l$. Each input client is to supply values for some of the variables, the others are to be chosen at random by the functionality. For each server P_i we define an index set D_i; these sets are mutually disjoint and designate subsets of the random inputs to $Q()$. If $j \in D_i$, the functionality will output x_j to P_i. Finally, the functionality will output $Q(x_1, ..., x_l)$ to the output clients, as well as Shamir shares of this value to the servers.

The functionality is specifically designed to fit into our protocol for computing Yao-garbled circuits to be presented later. In particular, some of the random values x_j will be used as encryption keys. Each such key has to be known to exactly one server, and this is the reason why the functionality outputs some of the x_j's to the servers. This functionality, denoted by $F_{Q,D_1,...,D_n}$, is more precisely defined as follows:

1. In the first round, it receives from each honest input client the x_j's this client supplies. In addition, it receives from the honest input clients and servers a set of values of the form produced in the linear preprocessing model. More precisely, these additional inputs take the following form:

   - For each x_j that is supplied by input client I, a set of values for $\mathsf{RandSS}^I(t)$ (as determined by a polynomial f_j).
   - For each x_j that is random, a set of values for $\mathsf{RandSS}(t)$ (as determined by a polynomial f_j).
   - For each x_j that is random and where $j \in D_i$, a set of values for $\mathsf{RandSS}^{P_i}(t)$ (as determined by a polynomial f_j).
   - A set of values for $\mathsf{RandSS}_0(dt)$ (as determined by a polynomial f_0).

   The functionality computes the shares and polynomials that all the above results in for the corrupt players and outputs this to the adversary. For instance, for every x_j supplied by corrupt input client I, this will be the polynomial $f_j()$. Note that, due to our assumed constraint on the number of corrupt servers, the honest players' information is enough to determine this information for the corrupt players. Also, for each x_j supplied by an honest input client, it sends $x_j - f_j(0)$ to the adversary. (No information is sent to honest players in this round.)

2. In round 2, the functionality receives from each corrupt input client the x_j's that it is responsible for. For each $j \in D_i$, the functionality will output x_j to P_i. It then outputs to each server a Shamir share of the value $Q(x_1, \ldots, x_l)$ generated by the polynomial $Q(f_1(), f_2(), ..., f_l()) + f_0()$ (this will be a univariate polynomial of degree at most dt). Finally, the functionality outputs $Q(x_1, \ldots, x_l)$ to all output clients.

We securely implement the above functionality in the linear preprocessing model using the following standard protocol:[6]

1. We do the following for each x_j: if x_j is supplied by input client I, execute $\mathsf{VSS}^I(t)$ where I uses x_j as his private input. The communication implied by this is the only communication in the first round.

   If x_j is random, execute $\mathsf{RandSS}(t)$. If x_j is random and $j \in D_i$, execute $\mathsf{RandSS}^{P_i}(t)$. In all cases, a set of shares of x_j is obtained. Let $x_{j,i}$ be the share of x_j obtained by server P_i. We execute $\mathsf{RandSS}_0(dt)$, creating shares of a degree dt polynomial that evaluates to 0 in 0. Let z_i be the share obtained by server P_i.

2. In the second round, each server P_i sends $Q(x_{1,i}, \ldots, x_{l,i}) + z_i$ to each output client. Each output client considers the values he receives as points on a degree dt polynomial f, reconstructs the polynomial (applying error-correction in the active adversary case) and outputs $f(0)$. Each server P_i outputs $Q(x_{1,i}, \ldots, x_{l,i}) + z_i$, and the values x_j for which $j \in D_i$.

We now show the security of this protocol using Canetti's UC framework [8]. We only show this for environments that supply inputs of correct form as specified above. This is sufficient, since

[6]For our purposes the degree d of Q will be no larger than 3, hence we will not consider optimizations that apply to a larger d.

we will only use the functionality in conjunction with the linear preprocessing, which is assumed to produce values of the right form.

Theorem 2.1 *There exists a 2-round protocol computing $F_{Q,D_1,\dots,D_n}$, the protocol is secure for all environments that supply inputs for honest players as specified in the description of $F_{Q,D_1,\dots,D_n}$. Furthermore, the protocol is secure for an adaptive adversary corrupting at most t servers and an arbitrary number of clients. For a passive adversary, we assume $dt < n$, for an active adversary we need $(d+2)t < n$. The communication complexity involves each output client receiving n field elements and each input client broadcasting its (masked) inputs.*

Proof: To prove security, we describe the required simulator (ideal model adversary), which as usual works by running an internal copy of the real-life adversary. In the first round, we receive a set of polynomials and shares from the ideal functionality, which we pass on to the adversary. In particular, this includes, for each corrupt input client, a random polynomial $f_j()$ of degree at most t, for each x_j this client supplies. When the adversary broadcasts a values r_j on behalf of the client, we compute $x_j = f_j(0) - r_j$ and give x_j as input to the ideal functionality. We also received from the ideal functionality $f_j(0) - x_j$ for each x_j supplied by an honest client. We use these values to simulate the broadcasts of honest clients. Note that for each corrupt server P_i, the information received from the functionality now defines a share $x_{j,i}$ of each x_j, and that it also defines a share z_i of the degree dt sharing of 0.

In the second round, the ideal functionality sends $Q(x_1, ..., x_l)$ to the simulator (we assume the adversary has corrupted at least one output client, otherwise the simulation becomes trivial). For each corrupt output client, we use the value $Q(x_1, ..., x_l)$ and the shares of it known by corrupt servers to interpolate a random polynomial f of degree at most dt with $f(0) = Q(x_1, ..., x_l)$ and $f(i) = Q(x_{1,i}, \dots, x_{l,i}) + z_i$, for each corrupt P_i, where the $x_{j,i}, z_i$ are the previously defined values for P_i. Then for each honest server P_i, we claim $f(i)$ as the value sent by P_i.

To establish adaptive security, we now show how to reconstruct the history of the players if they are corrupted after the protocol. Earlier corruptions are handled by truncating the reconstruction procedure.

If an input client or a server is corrupted after the protocol, we learn all his input, and pass this on to the adversary. This already determines his view of the protocol, and is consistent with what the adversary already knows by definition of the ideal functionality.

If an output client is corrupted after the protocol, we need not produce new values, as all output clients receive the same set of messages from honest servers, and these were already produced earlier.

It is straightforward to verify that this simulation leads to perfect indistinguishability between the real and ideal process. Namely, the only item that is produced using different algorithms in the two is the polynomial that determines $Q(x_1, ..., x_l)$. However, it is in both cases a random polynomial of degree at most dt under the constraints that it is consistent with corrupt servers' shares and its value at 0 is $Q(x_1, ..., x_l)$. ■

Note that if all terms in Q contain at least two inputs that are random, then the protocol can be modified to tolerate more corruptions in the passive case. Concretely, if a term contains the product of random variables x_1 and x_2, then the shares of these are available already in round 1. We can therefore compute a sharing of x_1x_2 using a polynomial of degree t which will be ready after round 1: we ask each P_i to compute $x_{1,i}x_{2,i}$, secret share this value with a degree t polynomial and send privately shares of it to all servers in round 1. Servers can now take an appropriate linear combination of the shares they receive to obtain shares of x_1x_2. We can then proceed as if we had a polynomial of degree $d-1$, treating x_1x_2 as one variable.

The protocol can be easily extended to secure computation in parallel of several low-degree polynomials, on some set of inputs, where some inputs may go to several polynomials. While it is not clear that this is implied by the composition theorem (because of the overlapping inputs) it can be shown by a trivial extension of the above proof. One can also modify the protocol so that only shares of $Q(x_1, ..., x_l)$ are computed for the servers, and not the value itself, by simply not sending the shares to the output clients.

The interest in the above results lies in the tradeoff between the bound on t and the round and communication complexity. Using standard techniques, say, from [5], it is straightforward to get an actively secure protocol for the same purpose with $O(\log d)$ rounds and where each server receives $O(dn^2)$ field elements, but where we only need to assume $n > 3t$.

3 Constant-Round MPC Using a Black-Box PRG

In this section we present our main protocol. We start in Section 3.1 by describing a variant of Yao's garbled circuit technique on which we rely. Then, in Section 3.2 we sketch the BMR approach for computing a garbled circuit in a distributed way. Finally, in Sections 3.3 and 3.4 we describe our modified approach.

3.1 The Basic Garbled Circuit Technique

Loosely speaking, the garbled circuit technique allows to represent a circuit C on ℓ input bits by an "encrypted" circuit $E(C)$ along with ℓ pairs of random keys, such that given $E(C)$ and the ℓ keys corresponding to a specific input $b_1, \ldots, b_\ell$, one can efficiently compute the output $C(b_1, \ldots, b_\ell)$, but this is the only information about the inputs that can be learned. Yao designed a method for generating such encrypted circuits, and used it to obtain a general constant-round two-party protocol for semi-honest parties. (See [30] for a formal proof of security of Yao's protocol and [32, 1] for other variants of this technique.) This protocol was generalized to the multi-party case by Beaver, Micali and Rogaway [4]. The circuit encryption process can be done in parallel for every gate in C, yielding a constant-round protocol for secure function evaluation.

We now describe how the basic technique for garbling a circuit works, by specifying how a trusted functionality could prepare an encrypted circuit and input keys as above. We will later show various protocols for implementing this functionality.

Without loss of generality, we assume the function to be computed is described as a Boolean circuit C with 2-input NAND gates. Let the number of wires in the circuit be W. We number the wires from 0 to $W-1$. For simplicity, we assume the circuit produces just a single output bit, to be learnt by all output clients, where this bit corresponds to the last wire, number $W-1$. Each input wire w has a bit b_w assigned to it, where each such bit is supplied by an input client. We will be using an index $0 \leq j \leq 2W-1$, where the values $2w, 2w+1$ are assigned to wire w.

We assume we have available a secure secret-key encryption scheme $E_S()$, where S is a k-bit key. We need to assume that the cryptosystem is semantically secure as long as each key is used on at most $2z$ messages, of length $k+1$ bits each, where z is the maximal fan-out in C.

To compute the garbled circuit, we get as input the bits b_w for each input wire, and then proceed as follows (see below for intuition):

- For every wire w, choose a random bit λ_w (masking the true value of the wire) and random keys S_{2w}, S_{2w+1}.
- For every gate g in C, do the following: suppose g has input wires α, β and output wire γ. Define the following values

$$\begin{array}{rclcrcl} a_g^{00} & = & S_{2\gamma+\delta_g^{00}} ; & \delta_g^{00} & = & (\lambda_\alpha \ nand \ \lambda_\beta) \oplus \lambda\gamma \\ a_g^{01} & = & S_{2\gamma+\delta_g^{01}} ; & \delta_g^{01} & = & (\lambda_\alpha \ nand \ \bar{\lambda_\beta}) \oplus \lambda\gamma \\ a_g^{10} & = & S_{2\gamma+\delta_g^{10}} ; & \delta_g^{10} & = & (\bar{\lambda_\alpha} \ nand \ \lambda_\beta) \oplus \lambda\gamma \\ a_g^{11} & = & S_{2\gamma+\delta_g^{11}} ; & \delta_g^{11} & = & (\bar{\lambda_\alpha} \ nand \ \bar{\lambda_\beta}) \oplus \lambda\gamma \end{array}$$

  Define $A_g^{cd} = (a_g^{cd}, \delta_g^{cd})$, for $c, d \in \{0, 1\}$. We compute the encryptions

$$\begin{array}{l} E_{S_{2\alpha}}(E_{S_{2\beta}}(A_g^{00})) \\ E_{S_{2\alpha}}(E_{S_{2\beta+1}}(A_g^{01})) \\ E_{S_{2\alpha+1}}(E_{S_{2\beta}}(A_g^{10})) \\ E_{S_{2\alpha+1}}(E_{S_{2\beta+1}}(A_g^{11})) \end{array}$$

- We output, for each g, the 4 encryptions as above along with the mask λ_{W-1} of the output wire - this is the encrypted circuit which we denote by $E(C)$. We also output, for each input wire w, the values $b_w \oplus \lambda_w$ and $S_{2w+(b_w \oplus \lambda_w)}$ - these are the encrypted inputs.

A word about the underlying intuition behind this: assume we knew all the inputs and did an ordinary computation of the circuit. This would result in assigning a bit b_w to every wire w. Instead, we get to know exactly one of the two encryption keys that are assigned to each wire, namely the key $S_{2w+(b_w \oplus \lambda_w)}$ and the bit $b_w \oplus \lambda_w$, and this is ensured for all input wires initially. We can think of this information as an encrypted representation of the bit b_w. This also means that by making λ_{W-1} public, we reveal the output bit, and only that bit.

The idea behind making the individual gates work in this scenario is to encrypt the keys and bits that might be assigned to the gate's output wire under keys assigned to input wires in such a way that players will be able to decrypt the "correct" key and bits for the output wire, and only this information. For instance, suppose that some gate g in the circuit has input wires α, β and output wire γ. If the information known for the input wires is $S_{2\alpha+c}, c$ and $S_{2\beta+d}, d$ for bits c, d, then the bit that should be revealed for output wire is $\delta_g^{cd} = ((c \oplus \lambda_\alpha) \ nand \ (d \oplus \lambda_\beta)) \oplus \lambda\gamma$, and so the key that should revealed is $S_{2\gamma+\delta_g^{cd}}$. The idea is therefore to encrypt these two values under $S_{2\alpha+c}$ and $S_{2\beta+d}$, for all 4 values of c, d.

Anyone who is given encrypted circuit and inputs can compute the output by the following *local circuit evaluation procedure*: for each input wire w, the key $S_{2w+(b_w \oplus \lambda_w)}$ and the bit $b_w \oplus \lambda_w$ are given. There will now be a number of gates, for which a key and a bit are known for both input wires. Let g be such a gate, say with input wires α, β and output wire γ. Since we know $b_\alpha \oplus \lambda_\alpha$ and $b_\beta \oplus \lambda_\beta$, we know which of the encryptions associated to g we can decrypt, namely those where both involved keys are known. We decrypt and obtain as result a key and a bit, which are easily seen to be $S_{2\gamma+(b_\gamma \oplus \lambda_\gamma)}$ and the bit $b_\gamma \oplus \lambda_\gamma$. Continuing this way, we will obtain a key and a bit $b_{W-1} \oplus \lambda_{W-1}$ for the output wire $W - 1$. Since we also know λ_{W-1}, we can compute the output bit b_{W-1}.

3.2 The BMR Protocol

The protocol from [4] can be seen as a concrete proposal for an encryption scheme E as required for the garbled circuit technique and a protocol for computing $E(C)$. Their scheme assumes a pseudorandom generator G taking as input a k-bit seed (such a generator can be constructed from any one-way function [21]). For a seed s, the output of $G(s)$ is split into k-bit blocks where the j'th block is denoted by $G(s)_j$.

A key S in the encryption scheme consists of n subkeys $S = (s^1, s^2, \dots, s^n)$, each of which is k bits long, and where initially s^i is known only to P_i. An element $m \in K$ is encrypted under S as $E_S(m) = m \oplus G(s^1)_j \oplus ... \oplus G(s^n)_j$, assuming m is the j'th k-bit string we encrypt under S.

Assuming m and each s^i have been secret shared among the servers, we can securely compute the encryption by having server i locally compute and secret share $G(s^i)_j$. We can then use linearity of the secret sharing to get shares of $E_S(m)$, and send these shares to the output clients. This will work, and makes only a black-box use of G, if the adversary is passive. But if he is active, each server needs to prove in zero-knowledge that he has computed and secret shared $G(s^i)_j$ correctly. In general, this requires generic zero-knowledge techniques, which means we no longer make a black-box use of G, and also leads to a major loss of efficiency.

3.3 Our Distributed Encryption Scheme

We now suggest a different encryption scheme for the garbled circuit technique, allowing to avoid the use of zero-knowledge proofs in the case of an active adversary. We assume as before a pseudorandom generator G which expands a k bit seed. A key S consists again of n subkeys $S = (s^1, s^2, \dots, s^n)$ where initially s^i is known only to P_i.

Consider now a situation where a message $m \in K$ has been secret shared among the n servers using a polynomial of degree d, where $t \leq d < n$. Let m_i denote the share of m given to P_i. To encrypt such a message under a key S, we will let each server encrypt the share he knows under his part of the key (expanded by G).

We define $E^j_S(m) = (G(s^1)_j \oplus m_1, ..., G(s^n)_j \oplus m_n)$. Having received the parts of the ciphertext $E^j_S(m)$ from the servers and given the key S, one can decrypt each share and reconstruct m from the shares, where error correction[7] is used to recover m if the adversary has actively corrupted some of the servers. The following lemma is straightforward.

Lemma 3.1 *The above distributed encryption scheme has the following properties:*

- *If an adversary is given up to t of the s^i's, and E^j_S is used on at most one message m, the encryption keeps m semantically secure.*
- *If the adversary is passive, and an honest output client is given S and receives $E^j_S(m)$ from the servers, he can decrypt correctly if $d < n$.*
- *If the adversary is active, and an honest output client is given S and receives $E^j_S(m)$ from the servers, he can decrypt correctly if $d + 2t < n$.*

This generalizes in a straightforward way to cases where two keys U, V are used. We write $E^i_U(E^j_V(m)) = (G(u^1)_i \oplus G(v^1)_j \oplus m_1, ..., G(u^n)_i \oplus G(v^n)_j \oplus m_n)$.

[7]In the case $t < n/2$ the subkeys and the message will be distributed using authenticated shares, in which case the decryption will involve a correction of *erasures* rather than errors.

In the following, we will be encrypting several elements in K under the same key. This is done in the natural way, by using a fresh part of the output from G for each new element.

3.4 Distributed Computation of a Garbled Circuit

We now apply the distributed encryption idea for securely computing a garbled circuit using a black-box PRG. The protocol takes place in the linear preprocessing model, and will use the subroutines and protocols described in Sections 2.1 and and 2.2.

1. In round 1, for each wire $w = 0..W-1$ the servers execute $\mathsf{RandSS}_{bin}(t)$ to create shares of the secret wire masks λ_w's. Also, for $i = 1..n, j = 0..2W-1$, they execute $\mathsf{RandSS}^{P_i}(t)$ to create shares of the subkeys s^i_j, such that s^i_j is known to P_i. Finally, for each input bit b_w held by input client I_j, the players execute $\mathsf{VSS}^{I_j}(t)$ (i.e., with I_j as dealer) and b_w as shared secret. Thus the only communication in round 1 consists of broadcasts done in the VSS subroutines.

2. In round 2, the servers first do some local computation.

   - For each input wire w and $i = 1..n$, each server locally computes a random share of the value $s^i_{2w+(b_w \oplus \lambda_w)}$. Note that since we work over a field of characteristic 2, this value can be written as a degree 2 polynomial, namely $(1 + b_w + \lambda_w)s^i_{2w} + (b_w + \lambda_w)s^i_{2w+1}$. We can therefore compute shares of the value $s^i_{2w+(b_w \oplus \lambda_w)}$ defined by a random degree $2t$ polynomial and send all these shares to the output clients, using the protocol from Section 2.2.
   - For each input wire w, the servers compute shares of the value $b_w \oplus \lambda_w$ and send them to the output clients.
   - For each gate g in the circuit, suppose the two inputs and output wire are wires α, β, γ, respectively. Then for each $i = 1..n$, the servers locally compute random shares of the values

$$\begin{array}{rclcrcl} a_g^{00,i} & = & s^i_{2\gamma+\delta_g^{00}} & ; & \delta_g^{00} & = & (\lambda_\alpha \; nand \; \lambda_\beta) \oplus \lambda\gamma \\ a_g^{01,i} & = & s^i_{2\gamma+\delta_g^{01}} & ; & \delta_g^{01} & = & (\lambda_\alpha \; nand \; \bar{\lambda_\beta}) \oplus \lambda\gamma \\ a_g^{10,i} & = & s^i_{2\gamma+\delta_g^{10}} & ; & \delta_g^{10} & = & (\bar{\lambda_\alpha} \; nand \; \lambda_\beta) \oplus \lambda\gamma \\ a_g^{11,i} & = & s^i_{2\gamma+\delta_g^{11}} & ; & \delta_g^{11} & = & (\bar{\lambda_\alpha} \; nand \; \bar{\lambda_\beta}) \oplus \lambda\gamma \end{array}$$

     Note that these values can be written as degree 3 polynomials in the already shared values, for instance, $a_g^{00,i} = (\lambda_\alpha\lambda_\beta + \lambda_\gamma)s^i_{2\gamma} + (1 + \lambda_\alpha\lambda_\beta + \lambda_\gamma)s^i_{2\gamma+1}$. We can therefore use the protocol from Section 2.2 to locally compute these random shares (without sending them to the output clients).

3. Let $a_g^{cd} = (a_g^{cd,1}, ..., a_g^{cd,n})$, for $c, d \in \{0, 1\}$. (This vector of n subkeys replaces the single key a_g^{cd} in the basic garbled circuit construction from Section 3.1). Define $A_g^{cd} = (A_g^{cd}, \delta_g^{cd})$. The servers can now reveal to the output clients encryptions of the form $E_{S_{2\alpha+c}}(E_{S_{2\beta+d}}(A_g^{cd}))$ using the distributed encryption scheme from Section 3.3. (Note that both the data we need to encrypt and the encryption subkeys are already shared in the required form.)

4. The output clients now apply the local circuit evaluation procedure described in Section 3.1, replacing ordinary decryption with distributed decryption (Section 3.3).

Theorem 3.2 (Black-box constant-round protocol in linear preprocessing model) *In the linear preprocessing model, there is a general 2-round MPC protocol making a black-box use of a pseudorandom generator. The protocol tolerates $t < n/2$ adaptively corrupted servers in the semi-honest case and $t < n/5$ in the malicious case. The communication complexity for computing a circuit C involves $O(n^2|C|k)$ bits sent to each output client, and each input client must broadcast its (masked) inputs to the n servers.*

Proof sketch: Formally speaking, we want to prove that the above protocol realizes a functionality F_C that accepts inputs $b_1, ..., b_\ell$ from the input clients and then outputs $C(b_1, ..., b_\ell)$ to all output clients.

Let $F_{low-degree}$ be an extended version of $F_{Q,D_1,...,D_n}$, computing all polynomials of degree $2, 3$ and shares we compute in the protocol π as described above (see the remarks following Theorem 2.1). Let $\pi^{F_{low-degree}}$ be the protocol we obtain by replacing in the natural way steps 1 and 2 in π by a call to $F_{low-degree}$. By Theorem 2.1 and the composition theorem, to show security of π, it is sufficient to show security of $\pi^{F_{low-degree}}$.

We now describe a black-box simulator for this protocol. The simulator proceeds by running internally copies of the linear preprocessing functionality, $F_{low-degree}$ and the (initially) honest players. The internal copies of honest players are called *virtual honest players.* They will be given 0's as input instead of the real values of their b_i's (which are unknown to the simulator). Otherwise all these internal entities proceed according to the protocol. There are only two differences between this and the real process: When the adversary specifies input bits to $F_{low-degree}$, this in particular fixes values of the b_i's for the corrupt players. The simulator then sends these bits to F_C. Second, when we get $C(b_1, ..., b_\ell)$ from F_C and construct the encrypted circuit, we assign a number of ciphertexts to the output gate g, exactly one of which will be decrypted in the local evaluation procedure. The simulator will put the bit $\lambda_{W-1} \oplus C(b_1, ..., b_\ell)$ as plaintext inside this encryption. This is done as follows: the degree $3t$ polynomial that defines this bit is of the form $g() + z()$ where $z()$ is a random degree $3t$ polynomial with $z(0) = 0$. We then change $z()$ to a random $z'()$of the same degree, but such that $g(0) + z'(0)$ is that value we want and $z'()$ is consistent with the shares of corrupt players. This change introduces no inconsistencies in the view of the virtual players, since $z()$ is used for nothing else than randomizing g. The simulated execution now results in output $C(b_1, ..., b_\ell)$ which is consistent with what the ideal functionality gives to the honest clients.

It remains to be described how the simulator handles corruptions. We describe how the simulator will reconstruct the view of a newly corrupted player. We assume the player is corrupted after the protocol terminates. For earlier corruptions, the reconstruction procedure is truncated appropriately. The general idea is that the simulator already has the views of the virtual honest players, including the one the adversary now wants to corrupt. We then modify this information so it becomes consistent with what we learn as a result of the corruption, without changing what the adversary already knows.

If an input client is corrupted, we learn his input bit(s), say b_w. We already broadcasted a value r_w related to this, and the virtual client has from the preprocessing a polynomial f_w with $f_w(0) = r_w$. We then change f_w to f'_w, so f'_w is random of degree at most t, subject to $f'_w(0) = b_w \oplus r_w$, and $f_w(c) = f'_w(c)$ for all corrupt P_c. We now want to claim that the virtual honest players used the polynomial $g'_w() = f'_w() + r_w$ in the further computation, instead of $g_w() = f_w() + r_w$ we used so far. Note that, to compute the garbled inputs, the protocol computes a set of degree-2 polynomials. Consider one of them, say $Q(b_w, \lambda_w, s_{2w}, s_{2w+1})$. Say the last 3 variables are shared using polynomials $g_2(), g_3(), g_4()$. Before corruption, we had $Q(0, \lambda_w, s_{2w}, s_{2w+1})$ shared using a univariate polynomial of the form $Q(g_w(), g_2(), g_3(), g_4()) + z_w()$, where $z_w()$ is random of degree $2t$ and $z_w(0) = 0$. Define $z'_w()$ by

$$Q(g_w(), g_2(), g_3(), g_4()) + z_w() = Q(g'_w(), g_2(), g_3(), g_4()) + z'_w()$$

and change $z_w()$ to $z'_w()$. This will change honest virtual server's shares, but not the corrupted server's shares, by construction of $f'_w(), g'_w()$. It will also preserve the data sent to output clients. We now give to the adversary the updated view of the virtual client.

If a server or an output client is corrupted, note that this does not result in any new data learnt from F_C. We can therefore give the current view of the virtual output client or server to the adversary.

This concludes the description of the simulator. The intuition of the analysis of the simulation is that all plaintext data are identically distributed in simulation as in real execution, the difference lies in the data that remain encrypted, and this cannot be detected efficiently by semantic security of the encryption.

Towards formalizing this: assume there exist environment and adversary for which simulation and real execution can be distinguished. We observe that the simulated execution can be changed into the real one by modifying the choice of a particular set of plaintexts residing inside undecrypted ciphertexts. We then define a series of hybrids by modifying one plaintext at a time from its value in simulation to its value in real life, and by a standard argument, there exists some hybrid that can be distinguished from its neighbor. Since each hybrid distribution can be sampled given the set of inputs we assume exists, this leads to an algorithm contradicting semantic security of the underlying encryption scheme. ■

From the remarks following Theorem 2.1, it follows that We can weaken the assumption on t in the active case to $t < n/3$, using standard techniques, at the expense of a larger (but still constant) number of rounds and more communication.

A word about how this result compares to earlier work: First, it was pointed out in [11] that MPC is possible in two rounds, given access to shares of random variables. This was argued using known general methods, and thus made a non-black-box use of the underlying PRG. Moreover, [11] only obtained security for $n > 3t$ in the passive case. A comparison to [4] was already made above.

3.5 The Plain Model

To implement the above protocol in the plain model, we need to emulate the procedures for generating random shared secrets. The cost of the resulting protocol is dominated by the cost of emulating $O(n|C|)$ invocations of (different variants of) $\mathsf{RandSS}(t)$. In the semi-honest case, each invocation of $\mathsf{RandSS}(t)$ can be implemented in a straightforward way by communicating $O(n^2)$ field elements: each player distributes a random secret, and the players output the sum of the shares they receive. (In fact, it suffices that $t + 1$ players share secrets.) Thus, the overall complexity in this case will be $O(n^3|C|k)$. In the malicious case, one could use a similar procedure based on any standard constant-round VSS protocol from the literature (e.g., the one from [5]). In fact, using the 2-round VSS protocol from [17], one can obtain a 3-round protocol in the plain model (assuming $t < n/5$).

We can weaken the assumption on t in the active case to $t < n/3$ by replacing the non-interactive polynomial evaluation protocol from Theorem 2.1 by an interactive one (e.g., using [5]). The resulting protocol will have a larger (but still constant) number of rounds and a higher communication complexity. In the appendix, we sketch how to use the VSS and multiplication protocol from [10] to further extend the feasibility result to the case $t < n/2$. This is based on two observations: first, a variant of our distributed encryption scheme can be used to encrypt values that have been shared under any VSS with a non-interactive reconstruction protocol. Second, by requiring that all values in the computation as well as shares of these values are VSS'ed, we can obtain a multiplication protocol that is guaranteed to terminate in a constant number of rounds, even for the case of $t < n/2$. Thus, our main feasibility result in the plain model is the following:

Theorem 3.3 (Black-box constant-round protocol in plain model) *In the plain model, there is a general constant-round MPC protocol making a black-box use of a pseudorandom generator. The protocol tolerates adaptively $t < n/2$ corrupted servers in the semi-honest case and $t < n/3$ in the malicious case.*

The above protocol is quite attractive when the number of servers is not too large. However, its extensive use of broadcasts makes the asymptotic communication complexity higher in the point-to-point model. To avoid paying this price, one can rely on an (information-theoretic) protocol of [22], which tolerates $t < n/3$ corrupt players and requires only $O(n^2k)$ amortized communication per gate (including "randomness" gates and "degree-2" gates). Since we only need to evaluate polynomials of a constant degree, the protocol of [22] can be implemented in a constant number of rounds. Thus, using [22], we get a protocol for computing a circuit C where $O(n^3|C|k)$ bits are exchanged between the servers, $O(n^2|C|k)$ bits are sent to each output client, and each input client must broadcast its (masked) inputs to the n servers.

References

[1] B. Applebaum, Y. Ishai, and E. Kushilevitz. Computationally private randomizing polynomials and their applications. In *Proc. 20th Conference on Computational Complexity*, 2005.

[2] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds. In *Proc. 8th ACM PODC*, pages 201–209, 1989.

[3] D. Beaver, J. Feigenbaum, J. Kilian, and P. Rogaway. Security with low communication overhead (extended abstract). In *Proc. of CRYPTO '90.*

[4] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *Proc. of 22nd STOC*, pages 503–513, 1990.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. of 20th STOC*, pages 1–10, 1988.

[6] C. Cachin, J. Camenisch, J. Kilian, and J. Muller. One-round secure computation and secure autonomous mobile agents. In *Proceedings of ICALP' 00*, 2000.

[7] R. Canetti. Security and composition of multiparty cryptographic protocols. In *J. of Cryptology*, 13(1), 2000.

[8] R. Canetti. Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. FOCS 2001: 136-145.

[9] R. Cramer and I. Damgård. Secure distributed linear algebra in a constant number of rounds. In *Proc. Crypto 2001.*

[10] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient Multiparty Computations Secure Against an Adaptive Adversary. In *Proc. EUROCRYPT* 1999, pages 311-326.

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

[12] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *Proc. of EUROCRYPT '00*, LNCS 1807, pp. 316-334, 2000.

[13] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Proc. of EUROCRYPT '01*, LNCS 2045, pp. 280-299, 2001.

[14] U. Feige, A. Fiat, and A. Shamir. Zero-Knowledge Proofs of Identity. J. Cryptology 1(2): 77-94 (1988).

[15] Uri Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *Proc. 26th STOC*, pages 554–563. ACM, 1994.

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

[17] R. Gennaro, Y. Ishai, E. Kushilevitz and T. Rabin. The Round Complexity of Verifiable Secret Sharing and Secure Multicast. In *Proceedings of the 33rd ACM Symp. on Theory of Computing (STOC '01)*, pages 580-589, 2001.

[18] R. Gennaro, Y. Ishai, E. Kushilevitz and T. Rabin. On 2-round secure multiparty computation. In *Proc. Crypto '02.*

[19] N. Gilboa and Y. Ishai. Compressing cryptographic resources. In *Proc. of CRYPTO '99.*

[20] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game (extended abstract). In *Proc. of 19th STOC*, pages 218–229, 1987.

[21] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[22] M. Hirt and U. M. Maurer. Robustness for Free in Unconditional Multi-party Computation. CRYPTO 2001: 101-118.

[23] Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *Proc. 41st FOCS*, pp. 294–304, 2000.

[24] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. Proceedings of 21st Annual ACM Symposium on the Theory of Computing, 1989, pp. 44 – 61.

[25] J. Katz and R. Ostrovsky. Round-Optimal Secure Two-Party Computation. In *CRYPTO 2004*, pages 335-354.

[26] J. Katz, R. Ostrovsky, and A. Smith. Round Efficiency of Multi-party Computation with a Dishonest Majority. In *EUROCRYPT 2003*, pages 578-595.

[27] J. Kilian. Founding cryptography on oblivious transfer. In *Proc. 20th STOC*, pages 20–31, 1988.

[28] Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. *J. Cryptology* 16(3): 143-184 (2003). Preliminary version in Crypto 2001.

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

[30] Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. Cryptology ePrint Archive, Report 2004/175, 2004.

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

[32] 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.

[33] R. Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. In *Proc. STOC 2004*, pages 232-241.

[34] R. Pass and A. Rosen. Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. FOCS 2003.

[35] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *Proc.* 21*st STOC*, pages 73–85. ACM, 1989.

[36] O. Reingold, L. Trevisan, and S. P. Vadhan. Notions of Reducibility between Cryptographic Primitives. *TCC 2004*: 1-20.

[37] P. Rogaway. *The Round Complexity of Secure Protocols.* PhD thesis, MIT, June 1991.

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

[39] S. R. Tate and K. Xu. On garbled circuits and constant round secure function evaluation. CoPS Lab Technical Report 2003-02, University of North Texas, 2003.

[40] A. C. Yao. How to generate and exchange secrets. In *Proc. 27th FOCS*, pp. 162–167, 1986.

A The Case $t < n/2$

In this section, we sketch how one can achieve a security threshold $t < n/2$ for an active, adaptive adversary.

We will use as point of departure the VSS and MPC protocols from [10], which were shown to be actively and adaptively secure. However, only the VSS protocol is constant-round.

In the VSS protocol, each player P_i receives a share a_i which is a standard Shamir share corresponding to a polynomial of degree at most t, and in addition some authentication information, of which one part is to be broadcast at reconstruction time together with a_i and the rest is used to verify information from other players at reconstruction time. Due to the non-linearity of the authentication information, this protocol does not entirely fit into our linear preprocessing model. Thus, in what follows we refer to the plain model.

We will use the VSS protocol such that input clients and servers may act as dealers, the secret is shared among the servers, and secrets can be reconstructed by servers and output clients. It is trivial to adapt the original protocol this way. The VSS protocol has a number of properties that we list here, with terminology adapted to the client/server model:

1. The reconstruction protocol is non-interactive: each sever broadcasts information, and each server and output client can then locally reconstruct the secret.

2. The VSS protocol is linear: if secrets a, b have been VSS'ed, then for known constants α, β, a VSS of $\alpha a + \beta b$ can be computed by servers locally computing the same linear combination of their shares and authentication information.

3. The VSS protocol allows a constant round multiplication protocol using a standard reduction to computation of linear functions: each server P_i holding shares a_i, b_i of a, b, can compute and VSS $a_i b_i$ and prove that this was correctly done. Then each server locally computes an appropriate linear combination of shares of all n VSS's, which results in a VSS of ab. Let $\lambda_1, ..., \lambda_n$ be the coefficients used here.

Note that the multiplication protocol assumes that all servers complete their part correctly. If this is not the case, we need to take a different action, this is the motivation for the extended VSS protocol we now describe.

The extended VSS, called EVSS, works as follows:

- The dealer D VSS's (in parallel) the secret a and t random field elements $c_1, ..., c_t$, and we think of $a, c_1, ..., c_t$ as coefficients of a polynomial $f()$ of degree at most t. Using linearity, compute locally VSS's of $f(i)$, for $i = 1, ..., n$. Finally open $f(i)$ privately to P_i: each server sends privately to P_i the information he would normally broadcast to reconstruct $f(i)$. The share of P_i is now $f(i)$ and the entire information from the VSS's of $a, c_1, ..., c_t$.

  When the EVSS is used as subprotocol later, some servers may have been disqualified because they have been found to be corrupt. In this case, $f(i)$ is publically reconstructed and kept by the remaining players as part of their share. If a server is later disqualified for any reason, his share will immediately be reconstructed and kept by the remaining players.

- To each EVSS will be attached a so called public correction value δ. The rule for reconstruction will be that we first reconstruct a value from the shares by interpolation, and then δ is added to get the final reconstructed value. When a dealer constructs an EVSS from scratch, δ is 0 by default, but for technical reasons, we will need non-zero values later.

The EVSS inherits the basic properties of the VSS: it has non-interactive reconstruction (we really only need to open the VSS of the secret itself), it is linear, and finally it allows a constant round multiplication protocol that cannot be blocked by the adversary:

1. Assume values a, b have been EVSS'ed, with public correction values δ_a, δ_b, and server P_i holds shares a_i, b_i of a, b. Note that this means that a_i, b_i are VSS'ed, and that $a_1, ..., a_n$ determine $a - \delta_a$ as standard Shamir shares. Now do the following in parallel for each P_i who has not been disqualified at this point:

   (a) P_i computes and VSS's $a_i b_i$ and proves he did this correctly. The protocol from [10] can be used directly here, as it works for any commitment scheme that is linear over the field used.

   (b) P_i extends the VSS of $a_i b_i$ to an EVSS by VSS-ing t random field elements thus determining a polynomial $f_i()$, and finally opening privately $f_i(j)$ to each P_j (each of these EVSS's have public correction value 0).

2. Each server P_i who failed to perform the above two steps correctly is disqualified, and we reconstruct publically a_i, b_i. Let C be the total set of disqualified servers. Note that if a player was disqualified for any reason external to the multiplication protocol, a_i, b_i was already public, so we can compute $\gamma = \sum_{P_i \in C} \lambda_i a_i b_i$.

3. Using linearity, we compute the linear combination of the EVSS's of $a_i b_i$ from $P_i \notin C$. Note that results in an EVSS of $(a - \delta_a)(b - \delta_b)$ with public correction value γ. ¿From this EVSS and the original ones for a, b, we can now use linearity to compute with no further interaction an EVSS of ab.

The final element we need is how to use our distributed encryption scheme with the EVSS. This is simple, since the scheme works with any VSS with non-interactive reconstruction: Suppose we want to encrypt value a that has been EVSS'ed, and server P_i holds a part s^i of the encryption key. P_i now computes all the information he would broadcast to reconstruct a, encrypts it under s^i, and sends the result to the output clients. If an output client knows the entire key, all contributions can be decrypted and the client can simulate the reconstruction protocol locally. If the encryption algorithm is semantically secure, and the key has been EVSS'ed, the adversary (computationally) learns nothing new from seeing an encryption.

This leads to the following constant-round protocol for general MPC:

1. The input clients EVSS their inputs among the servers. If a client fails to do this, we can use default values for his input.

2. In parallel, the servers create EVSS's of all random values needed to construct a garbled circuit for the desired computation (keys and wire masks). This is done in the standard way, by having each server EVSS his own random contribution and adding all contributions. If a server fails to do this, he is disqualified and his contributions are ignored.

3. As described earlier, all the values that need to be encrypted to form the garbled circuit and inputs can be computed by evaluating a number of constant degree polynomials on the inputs, keys and wire masks. We can therefore compute these values in EVSS'ed form in a constant number of rounds by running some instances of the multiplication protocol. Each server then performs his part of the encryptions and send the results to the output clients.

4. The output clients perform local evaluation of the garbled circuit as described earlier.

B The General Preprocessing Model

We now briefly consider the case of *general preprocessing*, in which the dealer is allowed to do any (polynomial time) computation and send the results to the players, as long as the computation is independent of the inputs to the actual computation to be done later. The main motivation for this model is to focus on how much work in MPC can be shifted to a preprocessing phase. Unfortunately, unlike the case of linear preprocessing, we do not have a non-interactive way of emulating the dealer in the general case of preprocessing.

Note first that the simplest way to construct an appropriate encryption scheme for Yao's technique is to build it directly from a pseudorandom bit generator, using only a single seed as key, and using a fresh part of the output as a 1-time pad on each new message to encrypt. In this way, the encrypted circuit has size $O(|C|k)$ bits.

This can be used to get a very efficient general protocol for the general preprocessing model. We define the preprocessing to output $E(C)$ to all players and for each input wire w, it outputs $S_{2w}, S_{2w+1}, \lambda_w$ to the input client that chooses input for this wire. For a passive adversary, it will then be sufficient that, once b_w is known, the input client sends $b_w \oplus \lambda_w, S_{2w+(b_w \oplus \lambda_w)}$ to each output client, who can then do the local circuit evaluation procedure.

For an active adversary, we need to make sure the output clients receive correct keys. One way to do this is to require the preprocessing to secret share S_{2w}, S_{2w+1} among the servers. In the computation phase, the input client broadcasts $b_w \oplus \lambda_w$, and the servers then send their shares of $S_{2w+(b_w \oplus \lambda_w)}$ to the output clients. This will directly allow reconstruction of the correct key if

$t < n/3$. But if we attach authentication tags to the shares, as in the protocol of Rabin and Ben-Or [35], we only need $t < n/2$. This assumes, of course, that clients receive keys in the preprocessing for verifying the tags. Summarizing, we have

Theorem B.1 *There exists general multiparty computation protocols for a Boolean circuit C in the general preprocessing model with the following properties. For a passive adversary, clients receive $O(|C|k)$ bits in the preprocessing, the computation phase is non-interactive and each output client receives $O(Ik)$ bits (where I is the number of inputs). For an active adaptive adversary, where $t < n/3$, clients receive $O(|C|k)$ bits in the preprocessing, the computation phase is 2 rounds, each input client must broadcast his (masked) input, and each output client receives a total of $O(nIk)$ bits from servers. If we only assume $t < n/2$, add $O(nIk)$ bits to the preprocessing data.*