Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings

Ivan Damgård, Claudio Orlandi, and Mark Simkin

Aarhus University, Aarhus, Denmark
{ivan, orlandi, simkin}@cs.au.dk

Abstract. We present a very simple yet very powerful idea for turning any semi-honestly secure MPC protocol into an actively secure one, at the price of reducing the threshold of tolerated corruptions. Our compiler leads to a very efficient MPC protocols for the important case of secure evaluation of arithmetic circuits over arbitrary rings (e.g., the natural case of $\mathbb{Z}_{2^\ell}$) for small number of parties. We show this by giving a concrete protocol in the preprocessing model for the popular setting with three parties and one corruption. This is the first protocol for secure computation over rings that achieves active security with constant overhead.

1 Introduction

Secure Computation. Secure Multiparty Computaton (MPC) allows a set of participants $P_1, \ldots, P_n$ with private inputs respectively $x_1, \ldots, x_n$ to learn the output of some public function f evaluated on their private inputs i.e., $z = f(x_1, \ldots, x_n)$ without having to reveal *any other information* about their inputs. Seminal MPC results from the 80s [Yao86, GMW87, BGW88, CCD88] have shown that with MPC it is possible to securely evaluate any boolean or arithmetic circuit with information theoretic security (under the assumption that a strict minority of the participants are corrupt) or with computational security (when no such honest majority can be assumed).

As is well known, the most efficient MPC protocols have only semi-honest security. What is perhaps less well known is that by settling for semi-honest security, we also get a wider range of domains over which we can do MPC. In addition to the standard approach of evaluating boolean circuits or arithmetic circuits over fields, we can also do computation efficiently over other rings. This has been demonstrated by the Sharemind suite of protocols [BLW08], which works over the ring $\mathbb{Z}_{2^\ell}$. Sharemind's success in practice is probably, to a large extent, due to the choice of the underlying ring, which closely matches the kind of ring CPUs naturally use. Closely matching an actual CPU architecture allows easier programming of algorithms for MPC, since programmers can reuse some of the tricks that CPUs use to do their work efficiently.

While semi-honest security is a meaningful security notion that is sometimes sufficient, one would of course like to have security against active attacks. However, the known techniques, such as the GMW compiler, for achieving active security incur a significant overhead, and while more efficient approaches exist, they need to assume that the computation is done over a field. Typically, such protocols, like the BeDOZa or SPDZ protocols [BDOZ11, DPSZ12, DKL$^+$13], start with a *preprocessing phase* which generates the necessary correlated randomness [IKM$^+$13] in the form of so called *multiplication triples* (using homomorphic encryption or OT). This is followed by an information theoretic and therefore very fast *online phase* where the triples are consumed to evaluate the arithmetic circuit. To get active security in the on-line phase, protocols employ information-theoretic MACs that allow to detect whether incorrect information is sent. These MACs need to be defined over a field to be secure, and in addition, for efficiency, the domain of the computation must be the same field or a subfield. This excludes, of course, the ring $\mathbb{Z}_{2^\ell}$. An incomparable alternative is to use garbled circuits. However, they incur a rather large overhead when active security is desired, and cannot be used at all if we want to do arithmetic computation directly over a large ring. Thus, a very natural question is:

Can we go from semi-honest to active security at a small cost and can we do so in a general way which allows us to do computations over general rings?

Our results. In this paper we address the above question by making three main contributions:

1. A generic transformation that compiles a protocol with semi-honest security against at least 2 corruptions into one that is actively secure (but against a smaller number of corruptions). This works both for the preprocessing and the standard model. The transformation preserves information theoretic security and for a constant number of parties it loses only a constant factor in efficiency. It can, for instance, be applied to (a variant of) the Sharemind protocol to get a 3-party protocol Π in the preprocessing model that is information theoretically secure against 1 corruption and obtains active security with abort.
2. A preprocessing protocol for 3 parties that generates multiplication triples to be used by Π as above. This preprocessing can generate triples over any ring $\mathbb{Z}_m$ and has constant computational overhead for large m, more precisely, if m is exponential in the statistical security parameter.
3. A generic transformation that works for a large class of protocols including most of those output by our semi-honest-to-active compiler. The transformation takes as input a protocol that is secure with abort and satisfies certain extra conditions. It produces a new protocol with complete fairness [CL14]. While security with abort means that the adversary gets to see the output and then can decide whether to force the protocol to abort, complete fairness means that the adversary must decide whether to abort without seeing the output. This is very relevant in applications where the adversary might "dislike" the result and would prefer that it is not delivered. The transformation introduces an additive overhead that only depends on the size of the output and not the size of the computation. It works in the honest majority without broadcast model. In this model we cannot guarantee termination in general (otherwise we could do broadcast from scratch which is impossible) so security with complete fairness seems to be the best we can hope for.

Discussion of results. Our semi-honest-to-active compiler is based on the idea of turning each party in the semi-honestly secure protocol into a "virtual" party, and then each virtual party is independently emulated by 2 or more of the real parties (i.e., each real party will locally run the code of the virtual party). Intuitively, if the number of virtual parties for which a corrupt party is an emulator is not larger than the privacy threshold of the original protocol, then our transform preserves the *privacy* guarantees of the original protocol. Further, if we can guarantee that each virtual party is emulated by at least one honest party, then this party can detect faulty behaviour by the other emulators and abort if needed, thus guaranteeing *correctness*. Moreover, if we set the parameters in a way that we are guaranteed an honest majority among the emulators, then we can even decide on the correct behaviour by majority vote and get full active security. While this in hindsight might seem like a simple idea, proving that it actually works in general requires us to take care of some technical issues relating, for instance, to handling the randomness and inputs of the virtual parties.

Clearly, the approach is closely related to replicated secret sharing which has been used for MPC before [Mau03, FLNW17] (see the related work section for further discussion). But to the best of our knowledge, this is the first general construction that transforms an entire semi-honestly secure protocol to active security. From this point of view, it can be seen as a construction that unifies and "explains" several earlier constructions.

While our construction works for any number of parties it unfortunately does not scale well, and the resulting protocol will only tolerate corruption of roughly $\sqrt{n}$ of the n parties and has a multiplicative overhead of order n compared to the semi-honest protocol. This is far from the constant fraction of corruptions we know can be tolerated with other techniques. We show two ways to improve this. First, while our main compiler preserves adaptive security, we also present an alternative construction that only works for static security but tolerates active $n/\log n$ corruptions, and has overhead $\log^2 n$. Second, we show that using results from [CDI+13], we get a protocol for any number n of parties tolerating roughly $n/4$ malicious corruptions. We do this by starting from a protocol for 5 parties tolerating 2 semi-honest corruptions, use our result to constructs a 5 party protocol tolerating 1 active corruption, and then use a generic comstruction from [CDI+13] based on monotone formulae. Note that a main motivation for the results from [CDI+13] was to introduce a new approach to the design of multiparty protocols. Namely, first design a protocol for a constant number of parties tolerating 1 active corruption, and then apply player emulation and monotone formulae to get actively secure multiparty protocols. From this point of view, adding our result extends their idea in an interesting way: using a generic transformation one can now get active and information theoretic security for a constant fraction of corruptions from a seemingly even simpler object: a protocol for a constant number of parties that is *semi-honestly* secure against 2 corruptions.

Our preprocessing protocol is based on the idea that we can quite easily make multiplication triples involving secret shared values $a, b, c \in \mathbb{Z}_m$ and where $ab = c \bmod m$ if parties behave honestly. The problem now is that the standard efficient approach to checking whether $ab = c \bmod m$ only works if m is prime, or at least has only large prime factors. We solve this by finding a way to embed the problem into a slightly larger field $\mathbb{Z}_p$ for a prime p. We can then check efficiently if $ab = c \bmod p$. In addition we make sure that a, b are small enough so that this implies $ab = c$ over the integers and hence also that $ab = c \bmod m$.

Combining this preprocessing for $m = 2^\ell$ with our semi-honest-to-active compiler, we get a new information theoretically secure replacement for the Sharemind protocol tolerating one active corruption of the 3 parties. The online part can be immediately plugged in, in place of the original protocol and is essentially as efficient.

Finally, our compiler for complete fairness (informally) works for protocols where the output is only revealed in the last round, as is typically the case for protocols based on secret sharing. Roughly speaking, the idea is to execute a general MPC protocol that guarantees security with abort up to its last round just before the outputs are delivered. We then compute verifiable secret sharings of the data that parties would send in the last round – as well as one bit that says whether sending these messages would cause an abort in the original protocol. Of course, this extra computation may abort, but if it does not and we are told that the verifiably shared messages are correct, then it is too late for the adversary to abort; as we assume an honest majority the shared messages can always be reconstructed. While this basic idea might seem simple, the proof is trickier than one might expect – as we need to be careful with the assumptions on the original protocol to avoid selective failure attacks.

1.1 Related Work

Besides what is already mentioned above, there are several other relevant works. Previous compilers, notably the GMW [GMW87] and the IPS compiler [IPS08, IPS09], allow to transform semi-honestly secure protocols into maliciously secure ones. The GMW compiler uses zero-knowledge proofs and, hence, is not black box in the underlying construction. It produces protocols which are far from practically efficient. The IPS compiler is blackbox in the underlying protocol, but requires oblivious transfer. It is unclear whether the IPS compiler can be used to produce practically efficient protocol).

In contrast, our compiler does not require any computational assumption and thus preserves any information theoretic guarantees the underlying protocol has. Our transform does not have any large hidden constants and can produce actively secure protocols with efficiency that may be of practical interest.

In a recent work by Furukawa et al. [FLNW17], a practically very efficient three party protocol with one active corruption was proposed. Their protocol uses replicated secret sharing and only works for bits. As the authors state themselves, it is not straightforward to generalize their protocol to more than three parties, while maintaining efficiency. In contrast, our protocol works over any arbitrary ring and can easily be generalized to any number of players. Furthermore our transform produces protocols with constant overhead, whereas their protocol does not have constant overhead.

The idea of using replication to detect active corruptions has been used before. For instance, Mohassel et al. [MRZ15] propose a three party version of Yao's protocol. In a nutshell, their approach is to let two parties garble a circuit separately and to let the third party check that the circuits are the same. The results in this work are more general in the sense that we propose a general transform to obtain actively secure protocols from semi-honestly secure ones.

The idea of using replication to obtain better security has been used previously. In [DK00], Desmedt and Kurosawa use replication to design a mix-net with t^2 servers secure against (roughly) t actively corrupted servers. A simple approach to MPC based on replicated secret sharing was proposed by Maurer in [Mau03]. It has been the basis for practical implementations like [BLW08].

2 Preliminaries

Notation. If $\mathcal{X}$ is a set, then $v \leftarrow \mathcal{X}$ means that v is a uniformly random value chosen from $\mathcal{X}$. When A is an algorithm, we write $v \leftarrow A(x)$ to denote a run of A on input x that produces output v. For $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For n party protocols, we will write P_{i+1} and implicitly assume a wrap-around of the party's index, i.e. $\mathrm{P}_{n+1} = \mathrm{P}_1$ and $\mathrm{P}_{1-1} = \mathrm{P}_n$. All logarithms are assumed to be base 2.

Security Definitions. We define semi-honest security with the real/ideal-world paradigm, where the corrupted parties can choose their inputs maliciously but follow the protocol execution honestly otherwise. This notion is sometimes known as augmented semi-honest security [Gol04] and is considered the right way of defining semi-honest behaviour [HL10]. Our definition explicitly captures protocols in the dealer model, where, initially, a trusted dealer distributes some correlated randomness. We model the dealer as an ideal functionality that takes no input and outputs correlated randomness to the parties. Protocols without a dealer are captured by our definitions by simply ignoring the ideal dealer functionality.

Definition 1 (Adaptive t-privacy in the semi-honest model). *Let $1 \leq t \leq n$. For all adaptive semi-honest PPT adversaries $\mathcal{A}$ that can corrupt up to t parties in the real world, there exists an ideal world adversary* Sim. *All parties corrupted by $\mathcal{A}$ follow the protocol execution honestly. Parties that are corrupted before the protocol execution, are allowed to choose their inputs arbitrarily. Parties that are corrupted during the protocol execution are not. Upon corruption of a party, the adversary gets that party's current view of the protocol execution.*

Let $\mathsf{Real}_{\Pi,A}$ be the view of the corrupted parties in a real execution of protocol Π. Sim *has access to an ideal functionality computing f and an ideal dealer functionality. Let $\mathsf{Ideal}_{\Pi,A}$ be the view that* Sim *outputs in the ideal world. We say Π realizes f with perfect t-privacy against static, semi-honest adversaries, if*

$$\{\mathsf{Ideal}_{\Pi,A}\}_{\lambda \in \mathbb{N}} \equiv \{\mathsf{Real}_{\Pi,A}\}_{\lambda \in \mathbb{N}}$$

For active security we will use the standard definitions for both adaptive active security with abort and adaptive active security with guaranteed output delivery. As a small reminder, in active security with guaranteed output delivery the protocol will always produce an output. If the adversary refuses to provide any input to the protocol for the corrupted parties, then their inputs are replaced by some default value. If the adversary stops at a later point in the protocol run, then the remaining honest parties can finish the protocol run without the corrupted parties.

Ideal Functionalities. The broadcast functionality $\mathcal{F}_{\mathsf{bcast}}$ (Figure 1) allows a party to send a value to a set of other parties, such that all receiving parties either receive the same value or abort by outputting $\perp$.

While unconditionally secure broadcast with termination among n parties requires that strictly less than $n/3$ parties are corrupted [PSL80], this bound does not apply to our broadcast with abort functionality.

A straightforward way to implement this functionality is to let the sender send its input to all receivers. All receivers echo the received value by sending it to all other receivers. Each receiver gets a value from the sender and one value from each other receiver. If all values match, then it outputs the value, otherwise it sends $\perp$ to all other parties. If a party receives $\perp$, and it has not yet sent $\perp$ to all other parties itself, then it sends $\perp$ to all parties, outputs $\perp$ itself, and terminates.

Using the coin flip functionality $\mathcal{F}_{\mathsf{cflip}}$ (Figure 2), a set of parties can jointly generate and agree on a uniformly random λ-bit string. In the case of an honest majority, this functionality can be implemented with information-theoretic security via verifiable secret sharing [CDN15] (note that in the case all parties will participate in the coin flip protocol, even if only some of the parties are supposed to learn the result of the coin flip).

Let $\mathbb{P}$ be the set of players that want to perform a coin flip. To realize the functionality, every participating party $\mathrm{P}_i \in \mathbb{P}$ secret shares a random bit string r_i among *all* the other players. Once every player in $\mathbb{P}$ shared its bit string r_i, we let all players in $\mathbb{P}$ reconstruct all bit strings and output $\bigoplus_i r_i$. This is done by having all players send all their shares to players in $\mathbb{P}$. Here we assume that reconstruction is non-interactive, i.e., players send shares to each other and each player locally computes the secret. Such verifiable secret sharing schemes exist, as is well known.

The standard functionality $\mathcal{F}_{\mathsf{triple}}$ (Figure 3) allows three parties P_1, P_2, and P_3 to generate a replicated secret sharing of multiplication triples. In this functionality, the adversary can corrupt one party and pick its shares. The remaining shares of the honest parties are chosen uniformly at random. The intuition behind this ideal functionality is that, even though the adversary can pick its own shares, it does not learn anything about the remaining shares, and hence it does not learn anything about the actual value of the multiplication triple that is secret shared. We will present a communication efficient implementation of this functionality in Section 5.

Functionality $\mathcal{F}_{\mathsf{bcast}}$

The ideal functionality runs with parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$ and adversary $\mathcal{A}$.

- P_i sends $(v, \mathbb{P})$ to $\mathcal{F}_{\mathsf{bcast}}$, where $v \in \{0,1\}^*$ and $\mathbb{P} \subset \{\mathrm{P}_1 \ldots \mathrm{P}_n\}$.
- If $\mathbb{P}$ contains a corrupted party, then $\mathcal{A}$ receives v and can decide whether to continue or to abort by sending a bit to the ideal functionality.
  - If $\mathcal{A}$ continues, then $\mathcal{F}_{\mathsf{bcast}}$ sends v to all $\mathrm{P}_j \in \mathbb{P}$.
  - If $\mathcal{A}$ aborts, then $\mathcal{F}_{\mathsf{bcast}}$ sends $\perp$ to all $\mathrm{P}_j \in \mathbb{P}$.

Fig. 1. The broadcast functionality

Functionality $\mathcal{F}_{\mathsf{cflip}}$

The ideal functionality runs with parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$ and adversary $\mathcal{A}$.

- The functionalities waits for messages of the form $(\mathsf{cflip}, \mathbb{P})$ from all parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$.
- When the functionality receives the same message $(\mathsf{cflip}, \mathbb{P})$ from all parties, the functionality $\mathcal{F}_{\mathsf{cflip}}$ picks a random string $r \leftarrow \{0,1\}^\lambda$ and outputs r to all parties in $\mathbb{P}$.

Fig. 2. The coin flip functionality

3 Our Semi-Honest to Active Security Transform

The goal of our transform is to take a semi-honest protocol and convert it into a protocol that is secure against a small number of active corruptions. For simplicity, let us start with a semi-honestly secure n-party protocol ($n \geq 3$) in the dealer model that we will convert into a protocol that is secure against *one* active corruption in the dealer model. In the dealer model we have a trusted party, the dealer, that can distribute some correlated information before the actual protocol execution happens. During the protocol execution the parties cannot communicate with the dealer, but can rely on the previously distributed information. The main challenge in achieving security against an actively corrupted party, is to prevent it from deviating from the protocol description and sending malformed messages. Our protocol transform is based upon the observation that, assuming one active corruption, every pair of parties contains at least one honest party. Now instead of letting the real parties directly run the semi-honest protocol, we will let pairs of real parties simulate virtual parties that will compute, using the semi-honest protocol, the desired functionality on behalf

Functionality $\mathcal{F}_{\mathsf{triple}}$

The ideal functionality is parameterized by an integer m, runs with parties P_1, P_2, P_3 and adversary $\mathcal{A}$.

- If the adversary $\mathcal{A}$ has corrupted a party P_i then the adversary can input $(\mathsf{corrupt}, v)$ where $v = (a_{i+1}, a_{i+2}, b_{i+1}, b_{i+2}, c_{i+1}, c_{i+2})$ all in $\mathbb{Z}_m$.
- Upon receiving init from all honest parties the functionality $\mathcal{F}_{\mathsf{triple}}$ picks the undefined (a_i, b_i, c_i) uniformly at random, such that $(a_1 + a_2 + a_3) \cdot (b_1 + b_2 + b_3) = (c_1 + c_2 + c_3) \in \mathbb{Z}_m$ and outputs:
  - (a_1, b_1, c_1) to P_2 and P_3,
  - (a_2, b_2, c_2) to P_3 and P_1,
  - (a_3, b_3, c_3) to P_1 and P_2.

Fig. 3. Triple generation functionality

of the real parties. More precisely, for $1 \leq i \leq n$, the real parties P_i and P_{i+1} will simulate virtual party $\mathbb{P}_i$. In the first phase of our protocol, P_i and P_{i+1} will agree on some common input and randomness that we will specify in a moment. In the second phase, the virtual parties will run a semi-honestly secure protocol on the previously agreed inputs and randomness. Whenever virtual party $\mathbb{P}_i$ sends a message to $\mathbb{P}_j$, we will realize this by letting P_i and P_{i+1} both send the same message to P_j and P_{j+1}. Note that when both P_i and P_{i+1} are honest, these two messages will be identical since they are constructed according to the same (semi-honest) protocol, using the same shared randomness and the previously received messages. The "action" of receiving a message at the virtual party $\mathbb{P}_j$ is emulated by having the real parties P_j and P_{j+1} both receive two messages each. Both parties now check locally whether the received messages are identical and, if not, broadcast an "abort" message. Otherwise they continue to execute the semi-honest protocol. The high-level idea behind this approach is that the adversary controlling one real party cannot send a malformed message and at the same time be consistent with the other honest real party simulating the same virtual party. Hence, either the adversary behaves honestly or the protocol will be aborted.

Remember that we need all real parties emulating the same virtual party to agree on a random tape and a common input. Agreeing on a random tape in the dealer model is simple, since we can just let the dealer pick a random tape for each virtual $\mathbb{P}_i$ and send it to the corresponding real parties P_i and P_{i+1}. In the process of agreeing on a random input for the virtual parties we need to be careful in not leaking any information about the real parties' original inputs. Towards this goal, we will let every real party secret share, e.g. XOR, its input among all virtual parties. Now instead of letting the underlying semi-honest protocol compute $f(x_1, \ldots, x_n)$, where real P_i holds input x_i, we will use it to compute $f'((x_1^1, \ldots, x_n^1), \ldots, (x_1^n, \ldots, x_n^n)) := f(\bigoplus_i x_1^i, \ldots, \bigoplus_i x_n^i)$, where virtual party $\mathbb{P}_i$ has input $\left(x_1^i, \ldots, x_n^i\right)$, i.e. one share of every original input.

As a small example, for the case of three parties, we would get $\mathbb{P}_1 = \{\mathrm{P}_1, \mathrm{P}_2\}$ holding input $\left(x_1^1, x_2^1, x_3^1\right)$, $\mathbb{P}_2 = \{\mathrm{P}_2, \mathrm{P}_3\}$ with input $\left(x_1^2, x_2^2, x_3^2\right)$, and $\mathbb{P}_3 = \{\mathrm{P}_3, \mathrm{P}_1\}$ with $\left(x_1^3, x_2^3, x_3^3\right)$. Since every real party only participates in the simulation of two virtual parties, no real party learns enough shares to reconstruct the other parties' inputs. More precisely, for arbitrary $n \geq 3$ and one corruption, each real party will participate in the simulation of two virtual parties, thus the underlying semi-honest protocol needs to be at least 2-private. Actually, each real party will learn not only two full views, but also one of the inputs of each other virtual party, since it knows the shares it distributed itself. As we will see in the security proof this is not a problem and 2-privacy is, for one active corruption, a sufficient condition on the underlying semi-honestly secure protocol.

The approach described above can be generalized to a larger number of corrupted parties. The main insight for one active corruption was that each set of two parties contains one honest party. For more than one corruption, we need to ensure that each set of parties of some arbitrary size contains at least one honest party that will send the correct message. Given n parties and t corruptions, each virtual party needs to be simulated by at least $t+1$ real parties. We let real parties $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$ simulate virtual party $\mathbb{P}_i$[1]. This means that every real party will participate in the simulation of $t+1$ virtual parties. Since we have t corruptions, the adversary can learn at most $t\,(t+1)$ views of virtual parties, which means that our underlying semi-honestly secure protocol needs to be at least $\left(t^2+t\right)$-private.

In the following formal description, let $\mathbb{P}_i$ be the virtual party that is simulated by $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$. When we say $\mathbb{P}_i$ sends a message to $\mathbb{P}_j$, we mean that each real party in $\mathbb{P}_i$ will send one message to every real party in $\mathbb{P}_j$. Let $\mathbb{V}_i$ be the set of virtual parties in whose simulation P_i participates.

Let f be the n-party functionality we want to compute, and $\Pi_{f'}$ be a $\left(t^2+t\right)$-private, semi-honestly secure protocol that computes f', i.e. it computes f on secret shares as described above. We construct $\tilde{\Pi}_f$ that computes f and is secure against t active corruption as follows:

The dealer:

1. Pick $r_i \leftarrow \{0,1\}^\lambda$ for all $i \in [n]$.
2. For each P_i, output $\{r_j | \mathbb{P}_j \in \mathbb{V}_i\}$ and everything the underlying dealer for $\Pi_{f'}$ would output for any virtual party in $\mathbb{V}_i$.

The protocol:

1. P_i splits its input x_i into n random shares, s.t. $x_i = \bigoplus_{1 \le j \le n} x_i^j$, and for all $j \in [n]$ send $(x_i^j, \mathbb{P}_j)$ to the broadcast ideal functionality.
2. P_i receives $\left(x_1^j, \ldots, x_n^j\right)$ for every $\mathbb{P}_j \in \mathbb{V}_i$ from the broadcast functionality. If any $x_i^j = \bot$, abort the protocol.
3. All virtual parties, simulated by the real parties, jointly execute $\Pi_{f'}$, where each real party in $\mathbb{P}_i$ uses the same randomness r_i that was distributed by the dealer. Whenever $\mathbb{P}_i$ receives a message from $\mathbb{P}_j$, each member of $\mathbb{P}_i$ checks that it received the same message from all parties in $\mathbb{P}_j$. If not, it aborts.

Theorem 1. *Let $n \ge 3$. Suppose $\Pi_{f'}$ realizes the n-party functionality f' with perfect correctness and $\left(t^2+t\right)$-privacy against adaptive, semi-honest adversaries in the dealer model. Then $\tilde{\Pi}_f$ as described above computes f in the $\mathcal{F}_{\mathsf{bcast}}$-hybrid dealer model and is maliciously secure against t adaptive active corruptions.*

Remark 1. As will be apparent from the proof below, the simulator that shows security of $\tilde{\Pi}_f$ never rewinds and makes only black-box use of the simulator for $\Pi_{f'}$. It therefore follows easily that if $\Pi_{f'}$ is UC secure, then so is $\tilde{\Pi}_f$. A similar remark applies to the results from the corollaries below.

Remark 2. In Step 1 of the protocol the parties perform a XOR based n-out-of-n secret sharing. We remark that any n-out-of-n secret sharing scheme could be used here instead. In particular, when combining the transform with the preprocessing protocol of Section 5, it will be more efficient to do the sharing in the ring $(\mathbb{Z}_m, +)$.

Remark 3. Our compiler is information-theoretically secure. This means that our compiler outputs a protocol that is computationally, statistically, or perfectly secure if the underlying protocol was respectively computationally, statistically, or perfectly secure. This is particularly interesting, since, to the best of our knowledge, our compiler is the first one to preserve statistical and perfect security of the underlying protocol.

Proof. Before getting into the details of the proof, let us first roughly outline the possibilities of an actively malicious adversary and our approach to simulating his view in the ideal world. The protocol can be split into two separate phases. First all real parties secret share their inputs among the virtual parties through the broadcast functionality. A malicious party P_i^* can pick an arbitrary input x_i, but the broadcast functionality ensures that all parties simulating some virtual party $\mathbb{P}_j$ will receive the same consistent share x_i^j from the adversary. Since every virtual party is simulated by at least one honest real party, the simulator will obtain

[1] Any other distribution of real party among virtual parties that ensures that each real party simulates equally many virtual parties would work as well.

all secret shares of all inputs belonging to $\mathcal{A}$. This allows the simulator to reconstruct these inputs and query the ideal functionality to retrieve $f(x'_1, \ldots, x'_n)$ where if P_j is honest then $x'_j = x_j$ is the input chosen by the environment and if P_j is corrupt $x'_j = \bigoplus_i x^i_j$ is the input extracted by the simulator. Having the inputs of all corrupted parties and the output from the ideal functionality, we can use the simulator of $\Pi_{f'}$ simulate the interaction with the adversary. At this point, there are two things to note.

First, we have n real parties that simulate n virtual parties. Since the adversary can corrupt at most t real parties, we simulate each virtual party by $t+1$ real parties. As each real party participates in the same amount of simulations of virtual parties, we get that each real party simulates $t+1$ virtual parties. This means that the adversary can learn at most t^2+t views of the virtual parties and, hence, since $\Pi_{f'}$ is $\left(t^2+t\right)$-private, the adversary cannot distinguish the simulated transcript from a real execution.

Second, the random tapes are honestly generated by the dealer. The simulator knows the exact messages that the corrupted parties should be sending and how to respond to them. Upon receiving an honest message from a corrupted party, the simulator responds according to underlying simulator. If the adversary tries to cheat, the simulator aborts. Aborting is fine, since, in a real world execution, the adversary would be sending a message, which is inconsistent with at least one other real party that simulates the same virtual party, and a receiving party would receive inconsistent messages.

Given this intuition, let us now proceed with the formal simulation. Let $\mathcal{A}$ be the adversary that corrupts at most t parties. Let $\mathbb{P}^*$ be the set of real parties that are corrupted before the protocol execution starts. Let $\mathbb{V}^*$ be the set of virtual parties that are simulated by at least one corrupt real party from $\mathbb{P}^*$. We will construct a simulator $\mathsf{Sim}_{\tilde{\Pi}_f}$ for $\mathcal{A}$ using the simulator $\mathsf{Sim}_{\Pi_{f'}}$ for f'.

$\mathsf{Sim}_{\tilde{\Pi}_f}$:

1. For each $\mathrm{P}^*_i \in \mathbb{P}^*$ and $j \in [n]$, the adversary $\mathcal{A}$ sends $(x^j_i, \mathbb{P}_j)$ to the broadcast functionality.
2. For each $\mathrm{P}^*_i \in \mathbb{P}^*$, compute $x_i = \bigoplus_j x^j_i$ and send it to the ideal functionality computing f to retrieve $z = f(x_1, \ldots, x_n)$, where all x_i with $i \notin \mathbb{P}^*$ are the honest parties inputs in the ideal execution.
3. For each $\mathbb{P}_j \in \mathbb{V}^*$, for each $\mathrm{P}_i \notin \mathbb{P}^*$ pick uniformly random x^j_i and send it to $\mathcal{A}$.

At this point, we know the inputs and random tapes of all currently corrupted parties. Since the messages the corrupted parties are supposed to send are a deterministic function thereof, we can always check whether the adversary follows the protocol. We continue our simulation and the interaction between $\mathcal{A}$ and $\mathsf{Sim}_{\Pi_{f'}}$ as follows.

– When $\mathsf{Sim}_{\Pi_{f'}}$ queries the ideal functionality for f', we return z.
– Whenever $\mathcal{A}$ outputs a message, check whether it is the expected message, and abort if not. Otherwise forward one of the (possibly) redundantly sent messages to $\mathsf{Sim}_{\Pi_{f'}}$.

Whenever the adversary adaptively corrupts a new party P_i, we go through all virtual parties $\mathbb{P}_j$ in $\mathbb{V}_i$ (the virtual parties simulated by P_i) and consider the following two cases. First, if $\mathbb{P}_j$ already contained a corrupted party, then we already know how to simulate the view for this virtual player. Second, if P_i is the first corrupted party in $\mathbb{P}_j$, then we instruct $\mathsf{Sim}_{\Pi_{f'}}$ to adaptively corrupt $\mathbb{P}_j$ and we forward the output (the current view of $\mathbb{P}_j$) of $\mathsf{Sim}_{\Pi_{f'}}$ to $\mathcal{A}$. Since the view of $\mathbb{P}_j$ contains this virtual party's random tape, we can continue our overall simulation as above.

The indistinguishability of $\mathsf{Sim}_{\tilde{\Pi}_f}$ from a real protocol execution follows directly from the indistinguishability guarantees of $\mathsf{Sim}_{\Pi_{f'}}$. The first phase (steps 1-3) of input share distribution is perfectly indistinguishable from a real protocol execution, since the adversary only sees uniformly random shares. In the second phase, we use the $\left(t^2+t\right)$-privacy of $\Pi_{f'}$. As we have already argued above, the adversary can at most learn t^2+t views of the virtual parties. In addition, the adversary also knows the input shares that it distributed to the other virtual parties that are simulated by exclusively honest real parties. Hence, strictly speaking it knows more than just the views of the t^2+t virtual parties. This, however, is not a problem.

Assume indistinguishability would not hold. This means there is a distinguisher, who chooses his inputs to the computation, that could distinguish a real transcript from one that is simulated by $\mathsf{Sim}_{\tilde{\Pi}_f}$. Using such a distinguisher we can directly construct a distinguisher, who breaks the privacy of the underlying semi-honestly secure for computing f', which leads to a contradiction.

□

Efficiency of our transform. In our transform every real party emulates $t+1$ virtual parties which constitutes the only computational overhead of our transform (if we ignore the computational effort in checking that the $t+1$ received messages are equal)

Since our transform mainly works by sending messages in a redundant fashion, it incurs a multiplicative bandwidth overhead that depends on the number of active corruptions we want to tolerate. Assume the underlying protocol $\Pi_{f'}$ sends a total of m messages and further assume that we want to tolerate t corruptions. This means that every virtual party $\mathbb{P}_i$ will be simulated by $t+1$ real parties. Whenever a virtual party $\mathbb{P}_i$ sends a message to $\mathbb{P}_j$, we send $(t+1) \cdot (t+1) = t^2 + 2t + 2$ real messages. Ignoring messages sent for the coin-flips and share distribution, our transform produces a protocol that sends at most $m \cdot \big(t^2 + 2t + 2\big)$ messages.

For the special case, where $n = 3$, $t = 1$, and $\mathbb{P}_1 = \{\mathrm{P}_1, \mathrm{P}_2\}$, $\mathbb{P}_2 = \{\mathrm{P}_2, \mathrm{P}_3\}$, and $\mathbb{P}_3 = \{\mathrm{P}_3, \mathrm{P}_1\}$, it holds that for all $i \neq j$, $|\mathbb{P}_i \cap \mathbb{P}_j| = 1$. Hence, every message from $\mathbb{P}_i$ to $\mathbb{P}_j$ is realized by sending 2 real messages, which results in $2m$ total messages sent during the second phase of our transform.

Active security without a dealer: Using $\mathcal{F}_{\mathsf{cflip}}$, we can make our blackbox transform applicable for protocols that do not have a dealer. Currently, the dealer distributes the random tapes for all the virtual parties. Instead, we let all parties that simulate some $\mathbb{P}_i$ agree on a common random tape using $\mathcal{F}_{\mathsf{cflip}}$ right after the input distribution phase. Since the construction and the proof of security stays almost the same, we will not state them here explicitly. It should be noted that for t corruptions we have $n \geq \big(t^2 + t\big) + 1$ and thus we are always in an honest majority setting. This means that we can implement our ideal coin flip functionality with information theoretic security via verifiable secret sharing (VSS) as described in Section 2.

Note that even though VSS in itself is powerful enough to realize secure multiparty computation, we only use it for the coin flip functionality. Thus, the number of VSSs we need depends only on the amount of randomness used in the semi-honest protocol, rather than the overall size of the desired computation. Besides (and perhaps more importantly) for a large class of semi-honest protocols we do not need VSS at all to compile them (see Corollary 3 below).

Let $\tilde{\Pi}_f$ denote the modified protocol as described above, then we get the following corollary.

Corollary 1. *Let $n \geq 3$. Suppose $\Pi_{f'}$ realizes the n-party functionality f' with perfect correctness and $\big(t^2 + t\big)$-privacy against adaptive semi-honest adversaries in the standalone model. Then $\tilde{\Pi}_f$ computes f in the $(\mathcal{F}_{\mathsf{bcast}}, \mathcal{F}_{\mathsf{cflip}})$-hybrid standalone model and is maliciously secure against t adaptive active corruptions.*

Guaranteed Output Delivery. At the cost of reducing the threshold t of active corruptions that our transform can tolerate, we can obtain guaranteed output delivery. For this we need to ensure that an adversary cannot abort in neither the first phase, nor the second phase of our protocol. In the first phase, when each real party broadcasts its input shares to the virtual parties, we can ensure termination by simply letting every $\mathbb{P}_i$ to be simulated by $3t + 1$ real parties. In this case each $\mathbb{P}_i$ contains less than $1/3$ corruptions and unconditionally secure broadcast (with termination) exists among the members of $\mathbb{P}_i$. Using this approach, the adversary can learn $t\,(3t + 1)$ views and thus the underlying protocol needs to be $\big(3t^2 + t\big)$-private.

Another approach that gives slightly better parameters is to only assume an honest majority in each $\mathbb{P}_i$ and use broadcast with abort. In this case the underlying protocol needs to be $\big(2t^2 + t\big)$-private and thus, since $n \geq \big(2t^2 + t\big) + 1$, unconditionally secure broadcast with termination exists among *all* parties. If a real party simulating a virtual party aborts during a broadcast (to members of $\mathbb{P}_i$), it will broadcast (with guaranteed termination) this abort to all parties. At this point an honest sender, who initiated the broadcast, can broadcast its share for that virtual party among all parties in the protocol. The intuition is that, since the broadcast failed, there is at least one corrupted party in the virtual party and thus the adversary already learned the sender's input share, so we do not need to keep it secret any more. If the sender is corrupt and does not broadcast its share after an abort, then all parties replace the sender's input by some default value.

In the second phase of our protocol, real parties simulating virtual parties are currently aborting as soon as they receive inconsistent messages, as they cannot distinguish a correct message from a malformed one. If we ensure that every virtual party is simulated by an honest majority, then, whenever a real party receives a set of messages representing a message from a virtual party, it makes a majority decision. That is, it considers the most frequent message as the correct one and continues the protocol based on this message. Let $\tilde{\Pi}_f$ denote the modified protocol as described above.

Corollary 2. *Let $n \geq 3$. Suppose $\Pi_{f'}$ realizes the n-party functionality f' with perfect correctness and $\left(2t^2 + t\right)$-privacy adaptive semi-honest adversaries in the standalone model. Then $\tilde{\Pi}_f$, as described above, computes f in the $(\mathcal{F}_{\mathsf{bcast}}, \mathcal{F}_{\mathsf{cflip}})$-hybrid standalone model, is maliciously secure against t adaptive active corruptions, and has guaranteed output delivery.*

From biased to active security. In the biased semi-honest security model, we still assume that all parties follow the protocol execution honestly, but corrupted parties have the additional power of choosing their random tapes in a non-adaptive, but arbitrary manner. Adversaries who behave honestly, but tamper with their random tapes have been previously considered in [MW16, AJL+12].

If our compiler starts with a protocol $\Pi_{f'}$ that is secure against biased semi-honest adversaries, then we can avoid the use of a coin-flipping functionality, since any random tape is secure to use. We can modify our compiler in a straightforward fashion. Rather than executing one coin-flip for every $\mathbb{P}_i$ to agree on a random tape, we simply let one party from each $\mathbb{P}_i$ broadcast an arbitrarily chosen random tape to the other members of $\mathbb{P}_i$. Let $\tilde{\Pi}_f$ denote the modified protocol as described above.

Corollary 3. *Let $n \geq 3$. Suppose $\Pi_{f'}$ realizes the n-party functionality f' with perfect correctness and $\left(t^2 + t\right)$-privacy against biased adaptive semi-honest adversaries in the standalone model. Then $\tilde{\Pi}_f$ computes f in the $\mathcal{F}_{\mathsf{bcast}}$-hybrid standalone model and is maliciously secure against t adaptive active corruptions.*

3.1 Tolerating more corruptions assuming static adversaries.

In this section we sketch a technique that allows to improve the number of corruptions tolerated by our compiler if we restrict the adversary to only perform static corruptions (i.e., the adversary must choose the corrupted parties before the protocol starts).

Instead of choosing which real parties will emulate each virtual party in a deterministic way (like in the compiler described in the previous section), we will now map real parties to virtual parties in a probabilistic way. Intuitively, since the adversary has to choose who to corrupt before the assignment (and the assignment is done in a random way), this can lead to better bounds when transforming protocols with a large number of parties.

Our new transform works as follows: At the onset of the protocol, the parties invoke $\mathcal{F}_{\mathsf{cflip}}$ and use the obtained randomness to select uniformly at random a set of real parties to emulate each virtual party. Then we execute the transformed protocol Π_f exactly as we specified above.

Let's define a virtual party in our transform to be *controlled by the adversary* if it is only emulated by corrupt real parties, and let's define a virtual party to be *observed by the adversary* if it is emulated by at least one corrupt real party. As in the proof of Theorem 1, we need two conditions for our transform to be secure: 1) no virtual party can be *controlled by the adversary*; and 2) the number of virtual parties *observed by the adversary* must be smaller than the privacy threshold of the semi-honest protocol $\Pi_{f'}$.

We now show that we can set the parameters of the protocol in a way that these two properties are satisfied (except with negligible probability) and in a way that produces better corruption bounds than our original transform.

In the analysis we assume that $n = \Theta(\lambda)$, where n is (as before) the number of virtual and real parties while λ is the statistical security parameter. We also assume that the privacy threshold of the underlying semi-honest protocol $\Pi_{f'}$ is cn for some constant c. Finally, let e be the number of real parties that emulate each virtual party, and let $e = u \log n$ for a constant u. The number of corrupt real parties that can be tolerated by our transform is then at most $d \cdot n/\log n$ for some constant d. We choose the constants d and u such that $c < 1 - du$.

To show 1), it is easy to see that (by a union bound) the probability that at least one virtual party is fully controlled by the adversary (i.e., it is emulated only by corrupt real parties) is at most:

$$n\left(\frac{dn}{n\log n}\right)^e = n\left(\frac{d}{\log n}\right)^e$$

Since we set $e = u\log n$, this probability is negligible.

As for 2), the probability that a virtual party is *not* observed by the adversary (i.e., it is emulated only by honest parties) is $(1 - d/\log n)^e$, so that the expected number of such parties is $n(1 - d/\log n)^e$ which for large n (and hence small values of $d/\log n$) converges to

$$n(1 - de/\log n) = n(1 - du).$$

As we choose d and u such that $c < 1 - du$, it then follows immediately from a Chernoff bound that the number of virtual parties with only honest emulators is at least cn with overwhelming probability. Let $\bar{\Pi}_f$ denote the protocol using this probabilistic emulation strategy. We then have:

Corollary 4. *Let $n = \Theta(\lambda)$. Suppose $\Pi_{f'}$ realizes the n-party functionality f' with cn-privacy against static semi-honest adversaries in the standalone model, for a constant c. Then $\bar{\Pi}_f$ computes f in the $(\mathcal{F}_{\mathsf{bcast}}, \mathcal{F}_{\mathsf{cflip}})$-hybrid standalone model and is maliciously secure against $d \cdot n/\log n$ static corruptions for a constant d.*

Moreover, compared to $\Pi_{f'}$, $\bar{\Pi}_f$ has a multiplicative overhead of $O((\log n)^2)$, which is asymptotically better than that of Π_f (the protocol obtained using our adaptively secure transform).

3.2 Achieving Constant Fraction Corruption Threshold

One drawback of the result from Theorem 1 is that the actively secure protocol we achieve does not scale very well with the number of parties, in that is only secure against corruption of approximately $\sqrt{n}$ of the n parties. This is of course far from the constant fraction of corruptions we know can be tolerated with other techniques.

However, in [CDI+13] Cohen et al. show how to construct multiparty protocols for any number of parties from a protocol for a constant number k of parties and a log-depth threshold formula of a certain form, namely the formula must contain no constants and consist only of threshold gates with k inputs that output 1 if at least j input bits are 1, where the given k-party protocol should be secure against $j - 1$ (active) corruptions. In [CDI+13], constructions are given for such formulae, and this results in multiparty protocols tolerating essentially a fraction $(j - 1)/(k - 1)$ corruptions.

For instance, from a protocol for 5 parties tolerating 2 semi-honest corruptions (in the model without preprocessing), our result constructs a 5 party protocol tolerating 1 active corruption. Applying the results from [CDI+13], we get a protocol for any number n of parties tolerating $n/4 - o(n)$ malicious corruptions. This protocol is maliciously secure with abort, but we can instead start from a protocol for 7 parties tolerating 3 semi-honest corruptions and use Corollary 2 to get a protocol for 7 parties, 1 active corruption and guaranteed output delivery. Applying again the results from [CDI+13], we get a protocol for any number n of parties tolerating $n/6 - o(n)$ malicious corruptions with guaranteed output delivery. These results also imply that if we accept that the protocol construction is not explicit, or we make a computational assumption, then we get threshold exactly $n/4$, respectively $n/6$.

4 Achieving Security with Complete Fairness

The security notion achieved by our previous results is active security with abort, namely the adversary gets to see the output and then decides whether the protocol should abort – assuming we want to tolerate the maximal number of corruptions the construction can handle. However, security with abort is often not very satisfactory: it is easy to imagine cases where the adversary may for some reason "dislike" the result and hence prefers that it is not delivered.

However, there is a second version that is stronger than active security with abort, yet weaker than full active security, which is called *active security with complete fairness* [CL14]. Here the adversary may tell the functionality to abort or ask for the output, but once the output is given, it will also be delivered to the honest parties.

In this section we show how to get general MPC with complete fairness from MPC with abort, with essentially the same efficiency. This will work if we have honest majority and if the given MPC protocol is what we call (for lack of a better name) *well-formed*, a condition that is satisfied by a large class of protocols. The skeptical reader may ask why such a result is interesting, since with honest majority we can get full active security without abort anyway. Note, however, that this is only possible if we assume that unconditionally secure broadcast with termination is given as an ideal functionality. In contrast, we do not need this assumption as our results above, e.g., Corollary 1, can produce well-formed protocols that only need broadcast with abort (which can be implemented from scratch) and our construction below that achieves complete fairness does not need broadcast with termination either.

We define the following:

Definition 2. *A protocol is said to be* well-formed *if it satisfies the following:*

- *It has statistical or perfect active security*[2] *with abort, works for $t < n/2$ active corruptions and delivers the same output to all parties.*
- *One can identify a particular round in the protocol, called the* output round*, that has properties as defined below. The rounds up to but not including the output round is called the* computation phase*.*
- *The adversary's view of the computation phase is independent of the honest party's input. More formally, we assume that the simulator for the protocol is straight-line (it does not rewind) and can simulate the protocol up to the output round without asking for the output.*
- *The total length of the messages sent in the output round depend only on the number of players, the size of the output and (perhaps) on the security parameter*[3]*. We use $d_{i,j}$ to denote the message sent from party i to party j in the output round.*
- *At the end of the computation phase, the adversary knows whether a given set of messages sent by corrupt parties in the output round will cause an abort. More formally, there is an efficiently computable Boolean function f_{abort} which takes as input the adversary's view v of the computation phase and messages $\boldsymbol{d} = \{d_{i,j} | \ 1 \leq i \leq t, 1 \leq j \leq n\}$, where we assume without loss of generality that the first t parties are corrupted. Now, when corrupt parties have state v and send $\boldsymbol{d}$ in the output round, then if If $f_{abort}(v, \boldsymbol{d}) = 0$ then the protocol terminates, otherwise it aborts, where both properties hold except with negligible probability.*
- *One can decide whether the protocol aborts based only on all messages sent in the output round* [4]*. More formally, we assume the function f_{abort} can also take as input messages $\boldsymbol{d}_{all} = \{d_{i,j} | \ 1 \leq i \leq t, 1 \leq j \leq n\}$. Then, if parties $P_1, ..., P_n$ send messages $\boldsymbol{d}_{all}$ in the output round and $f_{abort}(\boldsymbol{d}_{all}) = 1$, then the protocol terminates, otherwise it aborts (except with negligible probability).*

A typical example of a well-formed protocol is one that is based on secret-sharing: in the computation phase the parties obtain (verifiable) secret sharings of the outputs and then these are opened in the output phase. For instance, if one applies our transformation from Section 3 to a standard secret-sharing based and semi-honestly secure protocol, then one gets a well-formed protocol. More generally, it is straightforward to

[2] We believe that our results also extend to the computational case, but since we are in an honest majority setting, we only focus on statistical and perfect security.

[3] In particular, it does not depend on the size of the evaluated function

[4] This restriction is only for simplicity, our results extend to the more general case where termination also depends on some state information that parties keep private, as long as the size of this state only depends on the size of the output.

see that if one applies Corollary 1 to a well-formed semi-honest protocol, then the resulting protocol is also well-formed.

We can now show the following:

Theorem 2. *Assume we are given a compiler that constructs from the circuit for a function f a well-formed protocol Π_f for computing f, tolerating t active corruptions. Then we can construct a new compiler that constructs a protocol Π'_f for computing f that also tolerates t active corruptions, is also well-formed but has security with complete fairness. The complexity of Π'_f is larger than that of Π_f by an additive term that only depends on the number of players, the size of the outputs and the security parameter.*

Proof. Let $Deal$ be a probabilistic algorithm that on input a string s produces shares of s in a verifiable secret sharing scheme with perfect t-privacy and non-interactive reconstruction, we write $Deal(s) = (Deal_1(s), \ldots, Deal_n(s))$ where $Deal_i(s)$ is the $i'th$ share produced. For $t < n/2$ this is easily constructed, e.g., by first doing Shamir sharing with threshold t and then appending to each share unconditionally secure MACs that can be checked by the other parties. Such a scheme will reconstruct the correct secret except with negligible probability (statistical correctness) and has the extra property that given a secret s and an unqualified set of shares, we can efficiently compute a complete set $Deal(s)$ that is consistent with s and the shares we started from.

Now given function f, we construct the protocol Π'_f from Π_f as follows:

1. Run the computation phase of Π_f (where we abort if Π_f aborts) and let $\boldsymbol{d}_{all} = \{d_{i,j} | \, 1 \le i \le t, 1 \le j \le n\}$ denote the messages that parties would send in the output round of Π_f. Note that each party P_i can compute what he would send at this point.
2. Let f' be the following function: it takes as input strings $\boldsymbol{d}_{all} = \{d_{i,j} | \, 1 \le i \le t, 1 \le j \le n\}$. It computes $Deal(d_{i,j})$ for $1 \le i,j \le n$ and outputs to party P_l $Deal(d_{i,j})_l$. Finally, it outputs $f_{abort}(\boldsymbol{d}_{all})$ to all parties.
   Now we run $\Pi_{f'}$, where parties input the $d_{i,j}$'s they have just computed.
3. If $\Pi_{f'}$ aborts or outputs 0, we abort. Otherwise parties reconstruct each $d_{i,j}$ from $Deal(d_{i,j})$ (which we have from the previous step): each party P_l sends $Deal(d_{i,j})_l$ to P_j, for $1 \le i \le n$ (recall that P_j is the receiver of $d_{i,j}$), and parties apply the reconstruction algorithm of the VSS.
4. Finally parties complete protocol Π_f, assuming $\boldsymbol{d}_{all} = \{d_{i,j} | \, 1 \le i \le t, 1 \le j \le n\}$ were sent in the output round.

The claim on the complexity of Π'_f is clear, since Π_f is well-formed and steps 2 as well as 3 only depend on the size of the messages in the output round and not on the size of the total computation.

As for security, the idea is that just before the output phase of the original protocol, instead of sending the d_i's we use a secure computation $\Pi_{f'}$ to VSS them instead and also to check if they would cause an abort or not. This new computation may abort or tell everyone that the d_i's are bad, but the adversary already knew this by assumption of wellformedness of Π_f. So by privacy of the VSS, nothing is revealed by doing this. On the other hand, if there is no abort and we are told the d_i's are good, the adversary can no longer abort, as he cannot stop the reconstruction of the VSSs.

More formally, we construct a simulator T as follows:

1. First run the simulator S for Π_f up to the output round. Then run the simulator S' for $\Pi_{f'}$ where T also emulates the functionality f' . In particular, T can observe the inputs S' produces for f' on behalf of the corrupt parties, that is, messages $\boldsymbol{d} = \{d_{i,j} | \, 1 \le i \le t, 1 \le j \le n\}$ where we assume without loss of generality that the first t parties are corrupt.
2. Note that T now has the adversary's view v of the computation phase of Π_f (from S) and messages $\boldsymbol{d}$, so T computes $f_{abort}(v, \boldsymbol{d})$. by assumption on wellformedness, this bit equals the output from f', so we give this bit to S'. If S' signals an abort, T also aborts.
3. If S' completes without aborting, we ask for the output y from f which we pass to S and tell it that the corrupt parties sent $\boldsymbol{d}$. By wellformedness we know that this will not cause S to abort, so it will now produce a complete set of messages $\boldsymbol{d}_{all}$ (including messages from the honest parties) that will be consistent with y and $\boldsymbol{d}$.

4. Finally T exploits t-privacy of the VSS: during the run of $\Pi_{f'}$ t shares of each $Deal(d_{i,j})$ have been given to the adversary. T now completes each set of shares to be consistent with $d_{i,j}$, and sends the resulting shares on behalf of the honest parties in Π'_f.
5. Finally, we let S complete its simulation of the execution of Π_f after the output round (if anyhting is left).

It is clear that T does not abort after it asks for the output. Further the output of T working with f is statistically close to that of the real protocol. This follows easily from the corresponding properties of S and S' and statistical correctness of the VSS. □

The construction in Theorem 2 is quite natural, and works for a more general class of protocols than those produced by our main result, but we were unable to find it in the literature.

It should also be noted that when applying the construction to protocols produced by our main result, we can get a protocol that is much more efficient than in the general case. This is because the computation done by the function f' becomes quite simple: we just need a few VSSs and some secure equality checks.

5 Preprocessing Protocol

In the following we present a preprocessing protocol for 3 parties that tolerates one active corruption and creates replicated secret sharings of multiplication triples $c = a \cdot b \in \mathbb{Z}_m$. In particular, the communication of our protocol is only $O(\log m + \lambda)$ many bits per generated triple, meaning that the overhead for active security is a constant when m is exponential in the (statistical) security parameter.

Combining this preprocessing protocol with the circuit evaluation approach presented by Beaver [Bea92] and Corollary 1, we obtain a three party protocol that is secure against one active corruption and has, furthermore, *constant* online communication overhead for any value of m i.e., during the online phase, the communication overhead does not depend on λ.

5.1 The protocol

Before the formal description of the protocol, we provide a high-level overview of the used techniques: we first let the parties generate integer secret sharings of two random values in $\mathbb{Z}_m$. The resulting secret shares will have $\log m + \lambda$ bits, where the extra λ factor guarantees that they are statistically close to random. When receiving shares (here and at any later point during the protocol), the parties will check that the shares are of the expected size and abort otherwise. Afterwards, using the multiplication protocol based on replicated secret sharing, the three parties compute the product of the two random values generated above. The protocol has statistical *privacy* even in the presence of a corrupt party but, however, the *correctness* of the generated triple is not guaranteed yet. To do so, we let the parties generate a second multiplicative triple in $\mathbb{Z}_p$, where p needs to be sufficiently large, and use the standard technique of "sacrificing" one triple to check the other one. Note that this technique only works over a field, thus the prime modulo p.

We now proceed with a more formal description of the different parts of the protocol. We start by introducing some useful notation for replicated secret sharing:

Replicated Secret Sharing – Notation and Invariant: We write $[a]_{\mathbb{Z}} = (a_1, a_2, a_3)$ for an integer secret sharing of a and $[a]_p = (a_1, a_2, a_3)$ for a secret sharing modulo p. In both cases it holds that $a = a_1 + a_2 + a_3$ (where the additions are over the integer in the first case and modulo p in the latter). As an invariant for both kinds of secret sharing, each party P_i will know the shares a_{i+1} and a_{i-1}.

Replicated Secret Sharing – Input: When a party P_i wants to share a value $a \in \mathbb{Z}_p$, P_i picks uniformly random $a_1, a_2 \leftarrow \mathbb{Z}_p$ and defines $a_3 = a - a_1 - a_2 \mod p$. Then P_i sends shares a_{j-1} and a_{j+1} to P_j. Finally P_{i+1} and P_{i-1} echo a_i to each other and abort if the value they received in this echo phase differs from what they received from P_i. When using integer secret sharing instead, the shares need to be large enough to statistically hide the secret. That is, when a party P_i wants to share a value $a \in \{0, \ldots, m-1\}$, P_i picks uniformly random $a_1, a_2 \leftarrow \{0, \ldots, 2^{\lceil \log m \rceil + \lambda} - 1\}$ and defines $a_3 = a - a_1 - a_2$. Then P_i sends shares a_{j-1}

and a_{j+1} to P_j. Now, P_j checks if $|a_{j\pm1}| \leq 2^{\lceil \log m \rceil + \lambda + 1}$ and aborts otherwise.[5] Finally P_{i+1} and P_{i-1} echo a_i to each other and abort if the value they received in this echo phase differs from what they received from P_i.

Replicated Secret Sharing – Reveal: When parties want to open a share $[a]$, P_i sends its shares a_{i+1} and a_{i-1} to P_{i+1} and P_{i-1} respectively. When P_i receives share a_i from P_{i+1} and share a'_i from P_{i-1}, P_i aborts if $a_i \neq a'_i$ or outputs $a = a_1 + a_2 + a_3$ otherwise.[6]

Replicated Secret Sharing – Linear Combination: Since the secret sharing is linear we can compute linear functions without interaction i.e., when executing $[c] = [a] + [b]$ each party will locally add its shares. We consider three kind of additions:

- $[c]_p = [a]_p + [b]_p$, where all the shares are added modulo p;
- $[c]_{\mathbb{Z}} = [a]_{\mathbb{Z}} + [b]_{\mathbb{Z}}$, where the shares are added over the integers (note that the magnitude of the shares will increase when using integer secret sharing);
- $[c]_p = [a]_p + [b]_{\mathbb{Z}}$, where the shares are added modulo p. Note that in the this case, if a is uniform modulo p then c is uniform modulo p.[7]

Replicated Secret Sharing – Multiplication: Given two sharings $[a]_p$, $[b]_p$, we can compute a secret sharing of the product $[c = a \cdot b]$ in the following way:

1. P_i samples a random $s_i \leftarrow \mathbb{Z}_p$ and computes $u_i = a_{i+1}b_{i+1} + a_{i+1}b_{i-1} + a_{i-1}b_{i+1} + s_i$;
2. P_i sends u_i to P_{i+1} and s_i to party P_{i-1};
3. Finally, party P_i defines its own two shares of c as $c_{i+1} = u_{i-1} - s_i$ and $c_{i-1} = u_i - s_{i+1}$;

When performing multiplications with integer secret sharings, we need to ensure that the chosen randomness is large enough to hide the underlying secrets. In particular, given two sharings $[a]_{\mathbb{Z}}$, $[b]_{\mathbb{Z}}$, such that all shares are bounded by B, we can compute a secret sharing of the product $[c = a \cdot b]_{\mathbb{Z}}$ in the following way:

1. P_i samples a random $s_i \leftarrow \{0, \ldots, 2^{2\lceil \log B \rceil + \lambda + 2} - 1\}$ and computes $u_i = a_{i+1}b_{i+1} + a_{i+1}b_{i-1} + a_{i-1}b_{i+1} + s_i$;
2. P_i sends u_i to P_{i+1} and s_i to party P_{i-1};
3. P_i checks that the received shares are of the correct size i.e., $|u_{i-1}| \leq 2^{2\lceil \log B \rceil + \lambda + 3}$ and $|s_{i+1}| \leq 2^{2\lceil \log B \rceil + \lambda + 2}$
4. Finally, party P_i defines its own two shares of c as $c_{i+1} = u_{i-1} - s_i$ and $c_{i-1} = u_i - s_{i+1}$;

Armed with these tools we are now ready to describe our preprocessing protocol. The protocol is similar in spirit to previous protocols (e.g., [DO10, DPSZ12]) for generating multiplication triples, and like in previous work we start by generating two possibly incorrect triples, and then "sacrificing" one to check the correctness of the other. The main novelty of this protocol is that the two triples actually live in different domains (one is a an integer secret sharing, while the others is a modular secret sharing). For the sake of exposition we describe the protocol to generate a single multiplicative triple but, as with previous work, it will be more efficient to generate many triples in parallel.

The Preprocessing Protocol – Generate Random Triples:

1. Every P_i picks random $a_i, b_i \leftarrow \mathbb{Z}_m$ and generates sharings of $[a_i]_{\mathbb{Z}}, [b_i]_{\mathbb{Z}}$;
2. All parties jointly compute $[a]_{\mathbb{Z}} = [a_1]_{\mathbb{Z}} + [a_2]_{\mathbb{Z}} + [a_3]_{\mathbb{Z}}$ and $[b]_{\mathbb{Z}} = [b_1]_{\mathbb{Z}} + [b_2]_{\mathbb{Z}} + [b_3]_{\mathbb{Z}}$;[8]

[5] To keep the protocol symmetric, we use the bound for a_3 for all three shares.

[6] There is no need to explicitely check for the size of a share in the reconstruction phase since, by the assumption that at least one among P_{i+1} and P_{i-1} is honest, one of the received shares will be the correct one.

[7] We will use this property twice in the protocol: once, when mixing integer triples and p-modular triples in the multiplication checking phase, and finally, to argue that the resulting triples will be uniform modulo m.

[8] Note that if now we convert the sharing of $[a]_{\mathbb{Z}}$ to $[a]_m$ by having each party take their shares and locally reduce modulo m, we get that a is uniformly random in $\mathbb{Z}_m$ in the view of the adversary since at least one honest party choose a_i as a uniform value modulo m; the same argument applies symmetrically to $[b]_{\mathbb{Z}}$.

3. All parties jointly compute $[c]_{\mathbb{Z}} = [a]_{\mathbb{Z}} \cdot [b]_{\mathbb{Z}}$ (optimistically using the multiplication protocol described above);
4. Every P_i picks random $x_i, y_i, r_i \leftarrow \mathbb{Z}_p$ and generates sharings of $[x_i]_p, [y_i]_p, [r_i]_p$;
5. All parties jointly compute $[x]_p = [x_1]_p + [x_2]_p + [x_3]_p$ and $[y]_p = [y_1]_p + [y_2]_p + [y_3]_p$ and $[r]_p = [r_1]_p + [r_2]_p + [r_3]_p$;
6. All parties jointly compute $[z]_p = [x]_p \cdot [y]_p$ (optimistically using the multiplication protocol described above);
7. All parties open r;
8. All parties jointly compute $[e]_p = r[x]_p + [a]_{\mathbb{Z}}$;
9. All parties jointly open e, d, then compute and open
$$[t]_p = de - rd[x]_p - e[y]_p + r[z]_p - [c]_{\mathbb{Z}}$$
and abort if the result is not 0;
10. If the protocol did not abort, all parties output (modular) sharings $[a]_m, [b]_m, [c]_m$ by reducing their integer shares modulo m;

We now argue that:

Theorem 3. *The above protocol securely realizes $\mathcal{F}_{\mathsf{triple}}$ with statistical security parameter λ in the presence of one active corruption when $|p| = O(\log m + \lambda)$.*

Proof. We only give an informal argument for the security of the protocol, since its proof is quite similar to the proof of many previous protocols in the literature (such as [DO10, BDOZ11, DPSZ12], etc.).

We first argue for correctness of the protocol, focusing on steps 1,2 and 9: Note that, if there is an output, the output is correct and uniform modulo m. It is correct since, if $c = ab$ over the integer then $c = ab \mod m$ as well. And the values a, b, c are distributed uniformly since there is at least one honest party (in fact, two), who will pick a_i uniformly at random in $\mathbb{Z}_m$, therefore $a = a_1 + a_2 + a_3 \mod m$ will be uniform over $\mathbb{Z}_m$ as well (the same applies of course also to b and c).

We now describe the simulator strategy for the individual subroutines, and then we build the overall simulator for the protocol in a bottom-up fashion. To keep the notation simpler we assume that P_1 is corrupt. This is w.l.o.g. due to the symmetry of the protocol. To account for rushing adversaries, we always let the adversary send their message after seeing the message output by the simulator on behalf of the honest parties. As usual, the simulator keeps track of the shares that all parties (honest and corrupt) are supposed to hold at all times.

Simulator – Honest Parties Inputs: To simulate an honest party inputting a value a the simulator follows the share procedure but replacing a with 0. The simulator then sends a_2, a_3 to the adversary P_1 and stores a_1, a_2, a_3. Now the simulator receives a_2' (or a_3' depending on whether we are simulating a P_2 input or a P_3 input) back from the adversary and aborts if $a_2' \neq a_2$ (as an honest party would do).

When performing sharings modulo p, the distribution of the simulated a_2, a_3 are identical as in the real protocol (trivially for a_2, and since a_1 is random and unknown to the adversary, a_3 will be uniformly distributed in both cases). When performing integer sharings, the distribution of the simulated a_2 is trivially identical in the real and simulated execution while a_3 is statistically close. This can be easily seen considering the distribution of $a_3 + a_2$ which is $a - a_1$ in the real protocol and $-a_1$ in the simulated execution. Since $a < m$ and a_1 is uniform between 0 and $m \cdot 2^{\lambda}$ the distributions are statistically close with parameter λ.

Simulator – Corrupt Party Input: When simulating the input of the corrupt party P_1 the simulator receives (a_1, a_3) (on behalf of P_2) and (a_1', a_2) on behalf of P_3. The simulator aborts if $a_1 \neq a_1'$ (like the two honest party would do). When simulating an input in $\mathbb{Z}_p$ the simulator reconstructs $a = \sum_i a_i \mod p$. When simulating an integer input the simulator checks in addition that the shares received are of the right size and then reconstructs $a = \sum_i a_i$. Note that now $|a| < 3 \cdot 2^{\lceil \log m \rceil + \lambda + 1}$ which could be larger than m, but not larger than p given our parameters.

Simulator – Multiplication: When simulating multiplications the simulator picks random (u_3, s_2) (see below for the distribution) and sends them to P_1. Then the simulator receives (u_1, s_1) from P_1. This uniquely defines the corrupt party shares of c, namely $c_2 = u_3 - s_1$ and $c_3 = u_1 - s_2$. Note that the simulator can already now compute the error $\delta_c = c - ab$ from the stored shares of a, b and the received values u_1, s_1 i.e., $\delta_c = u_1 - (a_2b_2 + a_2b_3 + a_3b_2 + s_1)$. The simulator sets the final share of c to be $c_1 = ab + \delta_c - c_2 - c_3$ and remembers (c_1, c_2, c_3) and δ_c.

When simulating multiplications in $\mathbb{Z}_p$ the simulator picks (u_3, s_2) uniformly at random, thus the view of the adversary is perfectly indistinguishable in the real and simulated execution (trivially for s_2, and since in the real protocol s_3 is uniformly random and therefore u_3 is uniformly random as well).

When simulating integer multiplications the simulator picks (u_3, s_2) uniformly at random in $\{0, \ldots, 2^{2\lceil \log B \rceil + \lambda + 2} - 1\}$, thus the view of the adversary is statistically close in the real and simulated execution (trivially for s_2, and since in the protocol s_3 is used to mask a value of magnitude at most $3B^3$, the distributions are statistically close with parameter λ. Note that when simulating integer multiplications the simulator will also abort if the received shares (u_1, s_1) exceed their bounds. This means that at this point the value of $|c| = |\sum_i c_i|$ is bounded by $24B^2 2^\lambda$. As we know from the input phase that all shares are bound by $B = 2^{\lceil \log m \rceil + \lambda + 1}$ we get that by setting p to be e.g., larger than $100m^2 2^{2\lambda}$ we can ensure that even in the presence of a corrupt party the value of c will not exceed p.

Simulator – Fake Reveal: At any point the simulator can open a sharing (a_1, a_2, a_3) to any value $a + \delta_1$ of its choice. To do so, the simulator sends two identical shares $(a_1 + \delta_1)$ to P_1 (simulating that both the honest P_2 and P_3 send the same share to P_1). Then, P_1 sends its (possibly malicious) shares $a_2 + \delta_2$ and $a_3 + \delta_3$ to the simulator. Now the simulator aborts if $\delta_2 \neq 0$ or if $\delta_3 \neq 0$. Note the aborting condition is exactly the same as in the real protocol, where e.g., the honest P_2 receives a_2 from P_1 and a'_2 from P_3 and aborts if the two values are different. Finally note that the view of the adversary is exactly the same in the real and simulated execution.

Putting Things Together – Overall Simulator Strategy: We are now ready to describe the overall simulation strategy. Note that all the settings in which the simulator aborts in the previous subroutines are identical to the abort conditions of the honest parties in the protocol and moreover are "predictable" by the adversary (i.e., the adversary knows that sending a certain message will make the protocol abort).

0. As already described, the simulator keeps track of the shares that all parties (honest and corrupt) are supposed to hold at all times.

1a. (Send on behalf of P_2 and P_3) The simulator simulates P_2 and P_3 sharing values a_2, a_3, b_2, b_3 as described above (e.g., the input are set to be 0);

1b. (Receive from P_1) The simulator receives the (maliciously chosen) shares of a_1 using the procedure described above. In particular, now a_1 is well defined and bounded.

2. The simulator keeps track of the shares of a and b that all parties are supposed to store after the addition; (note that since the shares of the honest parties are simulated to 0 we have $a = a_1$ and $b = b_1$ at this point);
3. The simulator uses the simulation strategy for the multiplication protocol as explained above. If the simulation does not abort the value of c and δ_c are now well defined and bounded.
4. The simulator runs the sharing subroutine for $x_2, y_2, r_2, x_3, y_3, r_3$ (e.g., all values are set to 0).
5. The simulator keeps track of the shares of x, y and r that all parties are supposed to store after the addition; (at this point x, y and r are well defined);
6. The simulator uses the simulation strategy for the multiplication protocol as explained above. If the simulation does not abort the value of z and δ_z are now well defined.
7. The simulator now runs the fake reveal subroutine and opens r to a uniformly random value;
8. The simulator keeps track of the shares of e, d that all parties are supposed to store after the executions of the linear combination;

9a. The simulator runs the fake reveal subroutine and opens e, d to two uniformly random values; If the simulation did not abort so far the simulator runs the fake reveal subroutine and opens t to $r\delta_z - \delta_c \mod p$. The simulator aborts if $t \neq 0$ as an honest party do, but also aborts if $\delta_c \neq 0$ or $\delta_z \neq 0$.

10. If the simulator did not abort yet, then the simulator inputs the shares of the multiplicative triple owned by the adversary $(a_2, a_3, b_2, b_3, c_2, c_3)$ to the ideal functionality $\mathcal{F}_{\mathsf{triple}}$.

We have already argued for indistinguishability for the various subroutine (thanks to the large masks used in the integer secret sharings). Note that when we combine them in the overall simulator we add an extra aborting condition between a real world execution of the protocol and a simulated execution, namely that the simulation always aborts when the triple is incorrect (during the triple check phase). We conclude that the the view of the adversary in these two cases are statistically close in λ thanks to the correctness check at steps $4 - 8$: assume that the multiplication triples are correct i.e., that $z = xy \mod p$ and $c = ab$ over the integers. Now, if we make sure that p is large enough such that the shares of a,b, and c are the same over the integers and modulo p, then the resulting t will always be 0. Note that this is guaranteed by the check, during the sharing phase, of the magnitude of the shares chosen by the other parties. Finally, assume that $c \neq ab$ e.g., $c = ab + \delta_c$ (with $\delta_c \neq 0$) and $z = xy + \delta_z$.

Now the result of the check will be $t = r\delta_z - \delta_c \mod p$: Since the value r is picked by the simulator *after* the values δ_c, δ_z have already been defined, we finally have that t is equal to 0 with probability p^{-1} which is negligible as desired.

Acknowledgements: We thank the anonymous reviewers for their useful feedback. This project has received funding from: the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme (grant agreement No 669255); the Danish Independent Research Council under Grant-ID DFF6108-00169 (FoCC); the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA).

References

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008: 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206, Málaga, Spain, October 6–8, 2008. Springer, Heidelberg, Germany.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[CDI+13] Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron Rothblum. Efficient multiparty protocols via log-depth threshold formulae. *Electronic Colloquium on Computational Complexity (ECCC)*, 20:107, 2013.

[CDN15] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing.* Cambridge University Press, 2015.

[CL14] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 466–485, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[DK00] Yvo Desmedt and Kaoru Kurosawa. How to break a practical MIX and design a new one. In *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 557–572. Springer, 2000.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

[DO10] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 225–255, 2017.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004.

[HL10] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. Cryptology ePrint Archive, Report 2010/551, 2010. http://eprint.iacr.org/2010/551.

[IKM+13] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 600–620, Tokyo, Japan, March 3–6, 2013. Springer, Heidelberg, Germany.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 294–314. Springer, Heidelberg, Germany, March 15–17, 2009.

[Mau03] Ueli M. Maurer. Secure multi-party computation made simple (invited talk). In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02: 3rd International Conference on Security in Communication Networks*, volume 2576 of *Lecture Notes in Computer Science*, pages 14–28, Amalfi, Italy, September 12–13, 2003. Springer, Heidelberg, Germany.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 591–602, 2015.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.