Efficient MPC with a Mixed Adversary

Martin Hirt and Marta Mularczyk[⋆]

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland.
{hirt, mumarta}@inf.ethz.ch

Abstract. Over the past 20 years, the efficiency of secure multi-party protocols has been greatly improved. While the seminal protocols from the late 80's require a communication of $\Omega(n^6)$ field elements per multiplication among n parties, recent protocols offer linear communication complexity. This means that each party needs to communicate a constant number of field elements per multiplication, independent of n.

However, these efficient protocols only offer active security, which implies that at most $t < n/3$ (perfect security), respectively $t < n/2$ (statistical or computational security) parties may be corrupted. Higher corruption thresholds (i.e., $t \geq n/2$) can only be achieved with degraded security (unfair abort), where one single corrupted party can prevent honest parties from learning their outputs.

The aforementioned upper bounds ($t < n/3$ and $t < n/2$) have been circumvented by considering mixed adversaries (Fitzi et al., Crypto' 98), i.e., adversaries that corrupt, at the same time, some parties actively, some parties passively, and some parties in the fail-stop manner. It is possible, for example, to achieve perfect security even if 2/3 of the parties are faulty (three quarters of which may abort in the middle of the protocol, and a quarter may even arbitrarily misbehave). This setting is much better suited to many applications, where the crash of a party is more likely than a coordinated active attack.

Surprisingly, since the presentation of the feasibility result for the mixed setting, no progress has been made in terms of efficiency: the state-of-the-art protocol still requires a communication of $\Omega(n^6)$ field elements per multiplication.

In this paper, we present a perfectly-secure MPC protocol for the mixed setting with essentially the same efficiency as the best MPC protocols for the active-only setting. For the first time, this allows to tolerate faulty majorities, while still providing optimal efficiency. As a special case, this also results in the first fully-secure MPC protocol secure against any number of crashing parties, with optimal (i.e., linear in n) communication. We provide simulation-based proofs of our construction.

1 Introduction

In this work, we consider the problem of secure multi-party computation (MPC), where n mutually distrusted parties want to jointly perform some computation, represented by a circuit over a finite field, with input, output, multiplication, affine and randomness gates. We assume that each pair of parties is connected by a secure channel and that the communication is synchronous.

Intuitively, a protocol executed by the parties is secure if it is "as good as" an ideal trusted third party who performs the computation for the parties. This is formalized in the so-called real-world/ideal-world paradigm by requiring that anything an adversary can do in the real-world protocol execution can also be achieved in the ideal world with the trusted party. In this work, we focus on perfect security against a central, static adversary, who can corrupt up to t parties of its choice.

Corruption types. In the literature, several types of corruption are considered, in particular, active, passive (also called semi-honest) and fail-stop corruption[1]. Since the seminal works [BGW88, CCD88],

[⋆] Research supported by the Zurich Information Security and Privacy Center (ZISC).

[1] Here the adversary is only allowed to crash parties, and it does not learn their internal states. Sometimes this setting is relaxed by making the sending operation atomic — a party can only crash after sending all messages in a given round. Since this seems unrealistic, we do not consider this relaxation.

it is known that a perfectly-secure protocol can only tolerate $t < n/3$ active corruptions, or $t < n/2$ passive corruptions.[2] We note that one can tolerate $t < n$ active corruptions, at the cost of degrading security (i.e., give up on guaranteed output delivery), but in this work we focus on full security.

Fitzi et al. [FHM98] pointed out a trade-off between the adversary's capabilities and the overall number of corruptions. For example, considering passive adversaries has the advantage of being able to tolerate as many as $t < n/2$ corruptions (in the fail-stop setting this is even $t < n$). However, these weak corruption types are not practical — if a single party misbehaves, protocols for such settings give no security guarantees at all. On the other hand, considering only active corruptions may be too pessimistic, since it comes at the cost of lowering the threshold to $n/3$. [FHM98] showed that there is room in between the above "pure" settings and introduced the mixed-adversary setting, in which the adversary can simultaneously corrupt up to t_a parties actively, up to t_p parties passively, and up to t_f parties in a fail-stop manner.[3] Perfectly secure MPC in the mixed setting is possible if and only if $3t_a + 2t_p + t_f < n$.

The mixed setting strictly generalizes the pure settings and, in addition, offers flexibility, since it allows to trade off a few corruptions of a strong type for many corruptions of a weaker type. For example, decreasing t_a by one allows to tolerate three more fail-stop corruptions. A protocol for the mixed setting can tolerate, for example, $n/2$ fail-stop corrupted parties, in addition to slightly less than 1/6 actively corrupted parties (which gives dishonest majority).

Communication-Efficient MPC. The first protocols proving that MPC is possible in the pure settings [BGW88, CCD88] and in the mixed setting [FHM98] were quite inefficient, with communication costs of evaluating one multiplication gate as high as $\Omega(n^6)$ field elements. Since then, great improvements have been achieved in the pure settings [DI06, HN06, DN07, BH08, BFO12, GIP$^+$14, GLS19], leading to protocols with complexity linear in n. However, for the (from a practical viewpoint) more relevant mixed setting, no practically efficient protocols are known. Providing such protocols would immediately yield efficient solutions for all pure settings, and, additionally, all settings "in between".

1.1 Contributions

We present the first multi-party computation protocol with perfect security against mixed adversaries, and with the communication cost of computing one multiplication gate linear in the number of parties. We achieve the optimal resiliency of $3t_a + 2t_p + t_f < n$. Moreover, we provide simulation-based proofs of security of our constructions.

This immediately yields, as special cases, protocols with optimal communication complexity for all pure settings. In particular, we get a protocol that tolerates $t_f < n$ fail-corruptions, a realistic setting that has so far received surprisingly little attention in the MPC literature. Moreover, our result covers all settings in between, for instance, one with many fail-corruptions and few active corruptions, as long as $t_f + 3t_a < n$. We believe that considering such settings makes sense, since in many applications it may be more likely that a party crashes, rather than that it engages in a malicious, coordinated attack.

Perhaps surprisingly, many techniques used in the settings with only active or only passive corruptions fail in the setting with additional fail-corruptions. We briefly summarize a couple of problems:

– The efficient secret reconstruction protocol of [DN07] requires the number of correct parties (that is, neither actively nor fail-corrupted) to be in $\Omega(n)$. Intuitively, in order to reconstruct k sharings, these are expanded to n sharings (using an error-correcting code). These n sharings are then reconstructed (robustly), one sharing to one party, who then forwards the secret to all other parties. The error-correcting code allows to correct faults introduced by corrupted parties. In the active-only setting, error correction is possible as long as $k \leq n - 2t_a$, so one can reconstruct

[2] It is also possible to achieve higher thresholds, for example $t < n/2$ active corruptions, if one considers additional assumptions (such as a broadcast channel). In this paper we do not consider these settings.

[3] We stress that a fail-corrupted party can be simultaneously passively-corrupted, in which case it colludes with other corrupted parties.

$k = n/3$ sharings at costs $O(n^2)$. In the mixed setting, error-correction (with erasures) requires $k \leq n - 2t_a - t_f$, which might be as small as 1. So the quadratic costs cannot be amortized. To deal with this problem, we develop a more efficient, but non-robust, version of the protocol from [DN07], and employ an extended version of the player-elimination framework to make it robust.

– A standard technique for evaluating an input gate is to reconstruct an existing sharing of a random value towards the inputting party, who then broadcast the difference to its input. This approach breaks down in the mixed setting, if the inputting party crashes. This is because known broadcast protocols for the mixed setting [GP92] provide no guarantees for a corrupted sender, so the received value can be arbitrary (even if the sender is only fail-corrupted). In fact, a rushing adversary may even be able to change the sender's input to a related value.[4]
  We introduce a simple generic technique to deal with such situations — We execute the standard broadcast protocol that guarantees correctness only if the sender does not crash, and afterwards check if the party crashed, by executing a special heartbeat protocol. If so, its input is set to a default value (note that allowing this is unavoidable, e.g. if the party crashes before sending any messages).
– Hyper-invertible matrices [BH08] are used to generate big bunches of sharings of random values. Every party shares one input to the matrix. Then the parties verify the consistency of all outputs by reconstructing and verifying t_a of them. In the active setting, this is achieved by reconstructing $2t_a$ sharings (each to a different party). In the mixed setting one would need to reconstruct $2t_a + t_f$ sharings. This destroys the efficiency of the construction.
  To deal with this, we use the same approach as above — we change the semantics of the protocol from [BH08] and give guarantees analogous to those of the broadcast protocol. Specifically, correctness is guaranteed only if no party crashes. In case of crashes, inconsistent outputs can go through undetected — this will be handled in the extended player-elimination framework, using the heartbeat subroutine.
– In the player-elimination framework [HMP00], the actual computation is performed by a very efficient, so-called "detectable" protocol, that does not guarantee correctness, but that guarantees that an inconsistency is detected by a correct party. After the protocol, the parties agree on whether any inconsistencies were detected. If so, they identify the corrupted parties, eliminate them, and repeat the process. Several problems occur in the mixed setting. First, since we modified the semantics of hyper-invertible matrices, in our computation faults may not be observed by any correct party. Second, parties crashing during the identification process cause additional headache. Finally, it is not known how to identify a conflict between an honest and a corrupted party. In the active setting ($3t_a < n$), one can simply eliminate both of them, but in the mixed setting, eliminating a crashed party with an honest party would violate the threshold ($3t_a + 2t_p + t_f < n$).
  We deal with all these problems with a simple and elegant trick: We use the (almost) standard fault detection from [HMP00]. In case of crashes, we might identify wrong parties. So, once we know who should be eliminated, we check whether some party has crashed and, if so, we eliminate it (alone). Otherwise, the parties were identified correctly and can be eliminated.

We prove the security of our protocol in a simulation-based framework, which allows to simplify and modularize the proof. That is, for most of our subprotocols we define an ideal functionality and prove that it is realized by the construction. We can then use this idealized functionality in the higher-level protocol, and the security of the overall construction follows by the composition theorem. We note, however, that modularization of protocols executed within the player-elimination framework is nontrivial. Roughly, idealizing a detectable protocol makes it impossible to identify corrupted parties in case of a disruption (this usually involves publicly replaying the disrupted execution, and hence depends on the actual implementation of the protocol). Therefore, for some subprotocols, instead of defining ideal functionalities, we only prove that they satisfy certain properties, which are then used in further proofs.

[4] Note that the fail-corrupted parties do not reveal their inputs, unless they are also passively or actively corrupted.

1.2 Related Work

Communication-Efficient Active MPC. Since the seminal feasibility results [Yao82, GMW87, BGW88, CCD88, RB89, Bea90], the goal of reducing the communication complexity of actively-secure multi-party computation has received a lot of attention. The considerable (although polynomial) complexity of the first protocols was reduced to linear cost per multiplication gate in the cryptographic setting [DI06, HN06], in the unconditional setting (that is, allowing negligible error probability) [DN07, DIK10, BFO12, GIP$^+$14, GIP15], and in the setting with perfect security [BH08, GLS19].

There also exist efficient protocols that can tolerate a dishonest majority [DPSZ12, DKL$^+$13, KOS16, BHKL18] (where the online phase has linear complexity). We stress that such protocols can only offer degraded security [CL14], in particular, no fairness or guaranteed output delivery. In this work, we focus on full security.

On the negative side, it has been proved [DNPR16] that linear per-multiplication gate complexity is inherent for unconditionally secure protocols.

Communication Cost Independent of the Circuit Depth. Most protocols with linear communication complexity, including ours, communicate (over the whole execution) additional $O(D \cdot p(n))$ field elements, where D is the multiplicative depth of the circuit[5] and p is a small polynomial. This caveat was recently removed by Goyal et al. [GLS19], who construct a perfectly-secure protocol for the active setting with linear complexity and no dependency on the circuit depth. We expect that our techniques can be applied to the protocol of [GLS19], resulting in a more efficient (for certain circuits) protocol for the mixed setting. We leave proving this claim as an important open question.

Mixing Different Types of Corruptions. Protocols providing several guarantees, depending on the number and the type of corruptions, have been proposed [Cha90, IKLP06, Kat07]. These papers consider adversaries that *either* corrupt a number of parties passively, *or* corrupt a (smaller) number of parties actively. In contrast, in the mixed setting, an adversary *simultaneously* corrupts some parties passively and some actively.

The mixed setting was considered by Badrinarayanan et al. [BJMS18] in the context of round-efficient protocols. The authors present a constant-round protocol for the cryptographic setting, and assuming setup. They also give a simulation-based proof, in a model similar to ours. We note that their scheme is not communication-efficient.

The mixed setting has also been studied in the context of secure message transmission over an incomplete network [DDWY93, CPA$^+$08, APC$^+$08], and in the context of degraded security guarantees [HLMR11].

Ghodosi and Pieprzyk [GP09] strengthen the mixed-adversary setting without fail-stop corruptions (with $t_f = 0$) to the setting with a so-called omnipresent adversary. The mixed setting can also be generalized to deal with so-called general adversaries [BFH$^+$08, HMZ08] (we note that these general protocols are very inefficient in the threshold setting).

1.3 Protocol Overview

Our protocol follows the preprocessing model — it consists of two phases: the preparation (offline) phase, executed even before the inputs are specified, and the evaluation (online) phase, executed once the circuit and the inputs are known.

We employ circuit randomization [Bea92], which means that the goal of the preparation phase is to generate a number of shared multiplication triples and sharings of random values. This is done using hyper-invertible matrices [BH08], modified for the mixed setting. Then, in the evaluation phase, the circuit is evaluated gate by gate: input gates are evaluated with the help of existing sharings of random values, affine gates are evaluated locally using linearity of the secret sharing, and a bunch of

[5] The multiplicative depth of a circuit is the maximal number of multiplication gates on any path in the circuit from an input or random gate to an output gate.

multiplication gates is evaluated simultaneously with the help of multiplication triples (one per gate) and a new protocol for reconstructing a bunch of secrets towards all parties (using a non-robust version of the protocol from [DN07]). Evaluating an output gate corresponds to secret reconstruction.

Both phases employ the player-elimination framework [HMP00], modified for the mixed setting, where some parties are eliminated whenever the adversary disrupts the protocol. The eliminated parties are excluded from further computation, with the exception of providing inputs and receiving outputs. The preparation phase starts with the original party set, and the evaluation phase continues with the set resulting from the preparation phase. (Note that this is different than in [BH08], where only the preparation phase requires player elimination.)

The communication complexity of the preparation phase is $O(|C|n\kappa + n^3\kappa)$, where $|C|$ is the size of the circuit. The communication complexity of the evaluation phase is $O(|C|n\kappa + n^3\kappa + Dn^3\kappa + c_in^2\kappa)$, where D is the multiplicative and c_i is the number of input gates.[6]

1.4 Outline of the Paper

Section 2 presents the stand-alone simulation-based model with mixed adversaries. Section 3 contains some essential preliminaries; further preliminaries can be found in Appendix A. Section 4 considers byzantine agreement protocols needed in our setting. Section 5 extends the player-elimination framework to the mixed setting. Section 6 treats Shamir secret sharing and various reconstruction protocols used in this paper. Our main protocol is presented in Sections 7 (the preparation phase) and 8 (the evaluation phase). Finally, Section 9 contains conclusions and addresses universal composition.

2 Model

We consider a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of n parties, who want to compute a function, represented as a circuit over a finite field $\mathbb{F}$ with $|\mathbb{F}| \geq 2n$. The circuit contains c_i input, c_o output, c_m multiplication, c_a affine and c_r random gates.

Mixed adversaries. A mixed adversary can corrupt up to t_a parties actively, up to t_p parties passively, and up to t_f parties in a fail-stop manner. We denote by $\mathcal{P}_a, \mathcal{P}_p$ and $\mathcal{P}_f$ the sets of actually corrupted parties. The adversary gains full control over actively corrupted parties, and it sees the whole internal state of passively corrupted parties. Moreover, at any point in the protocol, it can make a fail-stop corrupted party crash. Once a party is crashed, it does not send any messages. The adversary cannot see the inputs nor the messages processed by fail-stop corrupted parties (unless they are simultaneously actively or passively corrupted). A party which is not actively corrupted and has not crashed yet is called *correct*.

Stand-alone Security. We consider the standard stand-alone model of [Can00] with static adversary and synchronous communication over perfectly-secure channels (for details of this model, see [Gol04] or [Can00]). We extend it to include mixed adversaries as follows. First, note that allowing multiple types of corruption simultaneously is straightforward. Passive and active corruptions are already modeled, so now we focus on fail corruptions.

Intuitively, we model fail corruptions as a very weak form of active corruptions, where the adversary (1) does not see the secret state or the messages received by a fail-corrupted party, and (2) only specifies the moment when such party stops sending messages in a given execution. Formally, the real-world execution starts with all fail-corrupted parties being correct. Then, each round proceeds as follows. First, the correct parties generate their messages, as in the protocol. The messages addressed to the actively- and passively-corrupted parties, together with the randomness used by the latter, are given to

[6] One can easily get linear dependency on the number of input gates, using the techniques of [BH08]. However, since the number of input gates is usually insignificant compared to the number of multiplication and affine gates, we do not include this optimization in this paper.

the adversary. She then specifies: (1) the messages sent by actively-corrupted parties, and (2) the set of fail-corrupted parties that crash in this round. For each party P_i crashed in this round, the adversary specifies a set of parties who still receive the message from P_i in this round. The crashed parties are no longer considered correct in this execution. Finally, all parties receive their messages.[7] The view of a party consists of its input, randomness and all received messages. The view of the adversary consists of the views of passively- and actively-corrupted parties.

In the ideal world, the ideal-world adversary (the simulator) interacts with the ideal functionality $\mathcal{F}$. As usual, he receives the inputs of passively-corrupted parties and modifies the inputs of the actively-corrupted ones. Moreover, for fail-corrupted parties, he chooses whether the input or a special symbol $\perp$ should be given to $\mathcal{F}$, where $\perp$ means that the party gives no input (note that this is inherent). In our model, fail-corrupted parties always receive outputs (this strong guarantee is achieved by our protocols).

Formally, we consider functionalities $\mathcal{F}$ with domain $(X \cup \{\perp\})^n$, where $\perp \notin X$. The semantic of $\perp$ is left to the functionality, e.g., it can be replaced by a default value. The ideal world is defined as follows: The parties that are not actively corrupted input values from X, and the simulator receives the inputs of passively-corrupted parties. He then chooses the inputs of actively-corrupted parties from $X \cup \{\perp\}$ and specifies a set D of fail-corrupted parties whose inputs are set to $\perp$. $\mathcal{F}$ is evaluated using the above inputs, and outputs are sent to all parties.

The standard security requirement is that for every (unbounded) real-world adversary $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$, such that the joint distribution of the view of $\mathcal{A}$ and the outputs of the correct parties executing the protocol is equal to the joint distribution of the output of $\mathcal{S}$ and the outputs of the correct parties computed by the ideal functionality $\mathcal{F}$.

Modular Composition. The hybrid world with fail corruptions is the standard one (note that fail corruptions do not affect ideal functionalities). Then, replacing ideal evaluation calls by subroutine calls is straightforward: a subroutine is called with the set of actually crashed parties reset (note that the set of fail-corrupted parties is constant). Observe that this means that crashed parties may "come back to life" in subroutines. This is a strong type of corruption, and our protocols are secure in this setting.[8]

Corruption-aware functionalities. The model of [Can00] was extended by Asharov and Lindell [AL17] to allow corruption-aware functionalities, whose code depends on the corrupted set. We generalize this to the mixed setting in the straightforward way. Specifically, our functionalities receive as inputs: (1) the inputs from the parties, and (2) the sets $\mathcal{P}_a, \mathcal{P}_p, \mathcal{P}_f$ of corrupted parties. Then, each functionality receives, upon initialization, the same set of parties actually controlled by the adversary. Note that this is well defined in the static setting. We refer to [Can00] for a formal description.

Finally, we note that the corruption-aware functionalities were used in [AL17] only as a tool to allow modular proofs. This is the same in our case. In particular, the final MPC functionality both in this paper and in [AL17] is not corruption aware.

3 Preliminaries

We assume some familiarity with Hyper-invertible matrices, circuit randomization and Shamir secret sharing. These concepts are also explained in Appendix A.

[7] In particular, this means that the crashed parties still receive their messages.

[8] An alternative solution would be to enforce a consistent set of crashed parties. However, modeling this would require techniques from the adaptive setting, such as the environment providing the crashed set. Our solution is simpler and cleaner.

3.1 Conventions

The current party set. In this paper we execute protocols within the player-elimination framework, where some parties are eliminated and the original party set $\mathcal{P}$ is reduced to a smaller set, which we denote by $\mathcal{P}'$. Accordingly, we use t'_a, t'_p, t'_f and n' to denote the corruption thresholds in $\mathcal{P}'$ and $|\mathcal{P}'|$, respectively. A tuple $(\mathcal{P}', t'_a, t'_p, t'_f)$ is called a *configuration*.

The current configuration is an input to most of our protocols. Security is guaranteed only if the inputted configuration is *valid*. Intuitively, we need that $3t'_a + 2t'_p + t'_f < n'$ holds in $\mathcal{P}'$. However, since we use Shamir sharings with degree $d = t_a + t_p$ (independent of $\mathcal{P}'$), we also require that such sharings are possible to reconstruct by the parties in $\mathcal{P}'$. This means that $t_a + t_p < n' - 2t'_a - t'_f$, which implies $3t'_a + 2t'_p + t'_f < n'$.

Definition 1. *A configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$ is* valid *if (1) $\mathcal{P}' \subseteq \mathcal{P}$, (2) $t_a + t_p < n' - 2t'_a - t'_f$, and (3) $|\mathcal{P}' \cap \mathcal{P}_{\mathsf{x}}| \leq t'_{\mathsf{x}}$ for all $\mathsf{x} \in \{a, p, f\}$, where $\mathcal{P}_{\mathsf{x}}$ denotes the actually corrupted parties.*

Complete break down. In some cases (which will never occur in the real execution) our functionalities provide no security at all. This happens, for example, when the parties input inconsistent sets $\mathcal{P}'$. In this case, we say that the functionality executes *Complete Break Down*, meaning that it immediately stops execution, sends all inputs to the adversary and allows her to set all outputs (see also [CDN12]). Simulation is trivial in this case, and we omit this part in the code of simulators.

Fixed matrices. At different times in the protocols we use hyper-invertible and Vandermonde matrices. We require that whenever in a protocol a matrix is chosen, all parties choose the same matrix. So, for example, a fixed matrix of maximal needed size can be chosen as the parameter of the protocol and the parties can use submatrices of appropriate sizes.

4 Byzantine Agreement

We consider two types of Byzantine-agreement protocols, namely consensus and broadcast. A consensus protocol allows a set of parties, each holding an input x_i, to reach agreement on a common value y, where $y = x$ if all correct parties have input $x_i = x$. The guarantees of consensus in the mixed setting were formalized by Garay and Perry [GP92]. They include consistency — the outputs of all parties that are not actively corrupted are equal, and persistence — if the inputs of all correct parties are equal to x, then the outputs are equal to x.[9]

A broadcast protocol allows a sender to publicly announce his value to all parties, where it is guaranteed that indeed all parties receive the same value. Broadcast can be trivially constructed from consensus: the sender sends his value to every party and then parties invoke consensus on the received values. Formally, in the mixed setting, we require the same consistency as in consensus — the outputs of all parties that are not actively corrupted are equal, and validity — if the sender stays correct until the end of the protocol and his input is x, then all parties output x.

A consensus protocol for the setting with active and fail-stop corruptions (that is, for the mixed setting with $t_p = 0$) was given by Garay and Perry [GP92]. Their protocol achieves one-bit consensus, assuming that $3t_a + t_f < n$. As the protocol is perfectly secure, it is also secure in the presence of (any number of) passively corrupted parties. The communication complexity of the protocol in [GP92] is $O(n^3)$. However, by applying the king-simulation technique of [BGP92], this complexity can easily be reduced to $O(n^2)$.

When we execute protocols within the player-elimination framework, Byzantine agreement is invoked among the parties in the reduced party set $\mathcal{P}'$, but all parties in $\mathcal{P}$ should learn the output.[10] Hence, we employ the following consensus protocol, where every party $P_i \in \mathcal{P}'$ has input x_i, and every

[9] In terms of [GP92], we require agreement and frangible validity. We do not consider strong validity in this paper.

[10] All parties must know the current state of the protocol.

party in $\mathcal{P}$ receives output. The protocol internally invokes a standard consensus protocol, and it is secure assuming that $3t'_a + 2t'_p + t'_f < n'$.

Protocol Consensus($\{x_i\}_{P_i \in \mathcal{P}'}$)

```
1: The parties in $\mathcal{P}'$ invoke a standard consensus protocol (for example, [GP92, BGP92]), where the input
   of $P_i$ is $x_i$.
2: Every party in $\mathcal{P}'$ sends the output to every party in $\mathcal{P} \setminus \mathcal{P}'$.
3: Every party in $\mathcal{P}'$ outputs the result of the consensus, and every party in $\mathcal{P} \setminus \mathcal{P}'$ determines its output
   using the majority rule.
```

Moreover, all parties in $\mathcal{P}$ should be able to act as a sender in broadcast.[11] Such broadcast can easily be constructed by having a sender $P_s \in \mathcal{P}$ send his value to all parties in $\mathcal{P}'$, who then invoke Consensus on the received values.

The cost of Consensus instantiated with the protocol of [GP92, BGP92] is $O(n^2)$ (note that n is the size of the initial set $\mathcal{P}$). We denote by $\mathcal{BA}(\kappa)$ the complexity of broadcasting or reaching consensus on a κ-bit message in the mixed setting. With the above protocol, $\mathcal{BA}(\kappa) = O(n^2\kappa)$.

Remark. Our broadcast protocol guarantees no validity in case a fail-corrupted sender crashes. This means that while the adversary can only prevent a fail-corrupted party from sending messages, she can modify the messages broadcasted by such party. In this paper we only consider such weak guarantees.

As a side remark, we note that an alternative solution would be to strengthen the broadcast. This can be done by invoking after the weak broadcast our Heartbeat protocol (defined in Section 5) on the sender. If Heartbeat succeeds (meaning that the sender is correct at the end), then the value from the first broadcast can be used. Otherwise (the sender crashed during or before Heartbeat), the broadcast fails and parties output $\perp$.

5 Player-Elimination Framework for the Mixed Setting

The player-elimination framework was introduced in [HMP00] with the goal of designing efficient actively secure multiparty protocols. The central idea is that every time the adversary actively disrupts the protocol execution, at least one malicious party is “eliminated”, and the disrupted part of the protocol is repeated.

The player-elimination framework reduces the problem of designing a resilient protocol to the problem of designing a *detectable* protocol for the same task. In a nutshell, a detectable protocol can give incorrect outputs, but only if this is noticed by at least one honest party. Privacy needs to be preserved even if the adversary disrupts the protocol execution. Since detectability is a much weaker requirement than resilience, detectable protocols for a given task are usually much more efficient. The player-elimination framework transforms a detectable protocol into a resilient one, essentially without complexity overhead.

A protocol executed within the player-elimination framework is evaluated in segments. The size of a segment can be arbitrary, but it influences the overall complexity. For each segment, four phases are executed: detectable computation, fault detection, fault localization and player elimination. In the first phase the parties evaluate the segment, using the (very efficient) detectable protocol. Then, in fault detection, they detect whether any of them noticed a disruption in the first phase. If this is *not* the case, they proceed to the next segment. Otherwise, in fault localization they localize a set of two disputing parties (that is, two parties such that each of them claims that the other is corrupted). Both these parties are eliminated in player elimination, and the current segment is repeated. The eliminated parties no longer take part in the computation, but they can still provide inputs and receive outputs.

[11] The reason is that eliminated parties can still provide inputs to the MPC protocol.

Unfortunately, the player-elimination framework for the active setting [HMP00] does not work in the mixed setting. One reason is that a detectable protocol only guarantees that a fault is noticed by a non-actively corrupted party. This means that if all parties who noticed it crash right after the protocol, the fault may not be detected. Moreover, fault detection and localization subprotocols for the active-only setting break down in the mixed setting. For example, the localized set of disputing parties may contain a crashed and an honest party. If such set was eliminated, the condition $3t_a + 2t_p + t_f < n$ would no longer hold.

We therefore enhance the player-elimination framework to cope with mixed adversaries as follows. First, we modify the notion of a detectable protocol. Roughly, if a party crashes during (or before) the execution of a detectable protocol, then the output can be incorrect even without a correct party noticing this. Accordingly, in fault detection the parties detect not only whether a correct party noticed a disruption, but also whether a party crashed. Then, in Fault Localization, two sets of parties are localized, such that unless a party in the first set has crashed, at least one of the two parties in the second set is actively corrupted. Afterwards, the protocol Heartbeat is invoked on each party in the first set, in order to determine whether it crashed. Finally, in player elimination the parties eliminate either the crashed parties in the first set or, if there are no such parties, all parties in the second set.

Protocol Convention. In the description of detectable protocols, we say that a (correct) party who notices a fault becomes *unhappy*. Formally, each party stores a binary value, which we call a happy-bit, and which can take values "happy" and "unhappy". When a party becomes unhappy, it sets its happy-bit to "unhappy". The state of the happy-bits is local to each party, but it is persistent between protocols. Furthermore, we adopt the convention that during a detectable protocol, an unhappy party does not send any messages.

Detectability in the Mixed Setting. We extend the definition of a detectable protocol of [HMP00] to the mixed setting. Informally, we require that a protocol may produce incorrect outputs, but only if this is noticed by a correct party, *or if any party crashed.* Moreover, we require completeness, that is, a protocol should give correct outputs if there are no faults during the execution. Furthermore, we require privacy, which must be preserved always, no matter if there were crashes or malicious behavior.

Definition 2. *A passively secure protocol Π for a set of parties $\mathcal{P}'$ is* detectable *if after the execution of Π at least one of the following is true: either (1) the outputs of all correct parties are correct, or (2) at least one correct party in $\mathcal{P}'$ is unhappy, or (3) at least one fail-stop corrupted party in $\mathcal{P}'$ crashed. Moreover, if all parties are honest, then no (correct) party becomes unhappy.*

Furthermore, Π guarantees privacy in all cases.

Fault Detection. The protocol FaultDetect is executed by the parties in $\mathcal{P}'$ during the phase of fault detection. Its goal is to unify the happy-bits of the parties, such that *all* parties become unhappy whenever any correct party is unhappy or any party crashed. We also require that FaultDetect is complete, by which we mean that at the end of FaultDetect all correct parties are happy whenever (1) all parties enter the protocol happy, (2) all parties behave according to the protocol specification, and (3) all fail-corrupted parties *finish* FaultDetect alive. In particular, this means that if conditions (1) and (2) are fulfilled, and no party crashed before FaultDetect, but *there was a crash during* FaultDetect, we do not put any requirements on the final values of the happy-bits, except that they are the same for all parties.

Protocol FaultDetect($\mathcal{P}', t'_a, t'_p, t'_f$)

```
1: Every $P_i \in \mathcal{P}'$ sends its happy-bit to every other party and sets its happy-bit to "unhappy" if from at
   least one party it does not receive a bit at all or if the bit it receives is "unhappy" (or if it was unhappy
   before).
```

```
2: The parties in $\mathcal{P}'$ run consensus on their happy-bits.
3: Every $P_i \in \mathcal{P}'$ sets its happy-bit to the result of the consensus.
```

Lemma 1. *Assuming that $3t'_a + t'_f < n'$, the protocol* FaultDetect *gives the following guarantees:*

(Consistency) *At the end of* FaultDetect *the values of the happy-bits are the same for all parties.*
(Correctness) *If any party starts* FaultDetect *being correct and unhappy, or if any party starts* FaultDetect *crashed, then at the end all parties are unhappy.*
(Completeness) *If all parties are honest and start* FaultDetect *happy, then at the end all parties are happy.*

The communication complexity of FaultDetect *is $O(n^2 + \mathcal{BA}(1))$.*

Proof. Consistency follows by consistency of consensus. For correctness, note that both an unhappy party and a party which starts FaultDetect crashed make every correct party set its happy-bit to "unhappy" in Step 1. By persistence of consensus, every correct party is unhappy at the end of FaultDetect. Finally, if there is no crashed party and no unhappy party at the start of FaultDetect, and if all parties behave according to the protocol specification and no party crashes, then the value of happy-bit at the beginning of consensus is "happy" for all correct parties. By persistence, every correct party is happy at the end of the protocol.

□

Heartbeat. The protocol Heartbeat allows the parties in $\mathcal{P}'$ to reach agreement on whether a given party $P_h \in \mathcal{P}'$ is alive. Formally, every party $P_i \in \mathcal{P}'$ outputs a binary value, equal to "alive" or "crashed".

Protocol Heartbeat$(P_h, (\mathcal{P}', t'_a, t'_p, t'_f))$

```
1: $P_h$ sends the value 1 to every party $P_j \in \mathcal{P}'$.
2: The parties invoke consensus, where the input of $P_j$ is 1 if it received the value 1 from $P_h$ and 0 otherwise.
   The parties output "alive" if the output of the consensus is 1 and "crashed" otherwise.
```

Lemma 2. *Assuming that $3t'_a + t'_f < n'$, the protocol* Heartbeat *gives the following guarantees:*

(Consistency) *All parties in $\mathcal{P}'$ output the same value "alive" or "crashed".*
(Correctness) *If P_h starts* Heartbeat *crashed, then the output of* Heartbeat *is "crashed".*
(Completeness) *If P_h finishes* Heartbeat *correct, then the output of* Heartbeat *is "alive".*

The communication complexity of Heartbeat *is $O(n + \mathcal{BA}(1))$.*

Proof. Consistency follows directly from consistency of consensus, while correctness and completeness follow from persistence of consensus. □

6 Secret Sharing

We employ the Shamir secret sharing [Sha79], where each party P_i has associated with it a unique value $\alpha_i \in \mathbb{F} \setminus \{0\}$. A value $s \in \mathbb{F}$ is correctly shared with degree d among the parties in $\mathcal{P}$ if every correct party $P_i \in \mathcal{P}$ holds a value $s_i \in \mathbb{F}$, such that all points (α_i, s_i) lie on a polynomial g of degree at most d with $g(0) = s$. A situation, in which s is d-shared among $\mathcal{P}$ is called a d-sharing of s and denoted by $[s]_d$. Sometimes we require a value to be simultaneously d-shared and d'-shared. Such a sharing is called a (d, d')-sharing of s and denoted $[s]_{d,d'}$. The Shamir secret sharing is linear, that is, affine operations on shared values can be performed directly on the respective shares, without any communication.

There are two protocols associated with a secret-sharing scheme: share and reconstruct. In this paper, we do not consider the share protocol, and use instead a protocol that generates sharings of random values (as explained in Section 7). On the other hand, we consider three reconstruction protocols, the guarantees of which we describe below. The details are given in Section B.

All reconstruction protocols take as input the sharing degree d and the current configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$.

– The private reconstruction protocol $\mathsf{RecPriv}(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$ *detectably* reconstructs the sharing $[s]_d$ towards $P_r \in \mathcal{P}$, assuming that $d' < n - t'_a$ and $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid. The communication cost is $O(n'\kappa)$.
– The private reconstruction protocol $\mathsf{RecPrivRobust}(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$ *robustly* reconstructs the sharing $[s]_d$ towards $P_r \in \mathcal{P}$, assuming that $d' < n - 2t'_a - t'_f$ and $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid. The communication cost is $O(n'\kappa)$.
– The public reconstruction $\mathsf{RecPub}(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f), [s_1]_d, \ldots, [s_\ell]_d)$ *detectably* reconstructs ℓ sharings towards all parties in $\mathcal{P}'$, assuming that $d' < n - t'_a$ and $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid. The communication cost is $O(\ell n'\kappa + n'^2\kappa)$. Technically, the protocol is a non-robust version of the protocol of [DN07] that reconstructs a bucket of sharings, using error correcting codes.

7 Preparation Phase

Recall that the goal of the preparation phase is to robustly generate a number of multiplication triples, as formally defined by $\mathcal{F}_{\mathsf{prepPhase}}$. To realize $\mathcal{F}_{\mathsf{prepPhase}}$, we proceed as follows. First, we define an intermediate functionality $\mathcal{F}_{\mathsf{triples}}$, which is essentially a non-robust version of $\mathcal{F}_{\mathsf{prepPhase}}$ (it also includes fault localization) and realize $\mathcal{F}_{\mathsf{prepPhase}}$ in the $\mathcal{F}_{\mathsf{triples}}$-hybrid model. The rest of this section is then devoted to realizing $\mathcal{F}_{\mathsf{triples}}$.

7.1 The Functionality $\mathcal{F}_{\mathsf{prepPhase}}$

The parties input to the functionality the desired number L of triples and the current configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$. Since the protocol employs player elimination, the resulting triples are shared among the parties in a smaller set $\mathcal{P}''$. The resulting configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ is outputted to all parties. Moreover, all parties in $\mathcal{P}''$ receive the shares of L triples, where the sharing degree is $d = t_a + t_p$.

We model the worst-case scenario, where the adversary is allowed to choose $(\mathcal{P}'', t''_a, t''_p, t''_f)$, as long as it is valid. She also decides on the shares of passively and actively corrupted parties.

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

The functionality receives sets of corrupted parties $\mathcal{P}_a, \mathcal{P}_p, \mathcal{P}_f$. Let $d = t_a + t_p$.

1: Receive from each party a number L and a valid configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$.
2: If these values (ignoring $\perp$) are not consistent among parties, or if $(\mathcal{P}', t'_a, t'_p, t'_f)$ is not valid, execute Complete Break Down. Else, send $(\textsc{Ok}, L, \mathcal{P}', t'_a, t'_p, t'_f)$ to the adversary and proceed as follows.
3: The adversary sends:
 – A valid set $\mathcal{P}'' \subseteq \mathcal{P}'$ with thresholds t''_a, t''_p, t''_f.
 – For each $P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)$, shares $((a_j^{(k)}, b_j^{(k)}, c_j^{(k)}))_{k=1\ldots L}$.
 (Set $\mathcal{P}'' = \mathcal{P}'$, $t''_{\mathsf{x}} = t'_{\mathsf{x}}$ and $(a_j^{(k)}, b_j^{(k)}, c_j^{(k)}) = (0,0,0)$ if valid values are not received.)
4: Send $(\mathcal{P}'', t''_a, t''_p, t''_f)$ to all parties.
5: Generate L triples shared among the parties in $\mathcal{P}''$ as follows. For each triple k, choose random $a^{(k)}$ and $b^{(k)}$, and let $c^{(k)} = a^{(k)}b^{(k)}$. Then, for each $k = 1, \ldots, L$ and $\mathsf{x} \in \{a, b, c\}$, choose the polynomial $g_{\mathsf{x}}^{(k)}$ at random from the set of all polynomials of degree at most d, going through the point $(0, \mathsf{x}^{(k)})$ and all points in $\{(\alpha_j, \mathsf{x}_j^{(k)}) \mid P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)\}$. Send to each $P_i \in \mathcal{P}''$ its shares $(g_a^{(k)}(\alpha_i), g_b^{(k)}(\alpha_i), g_c^{(k)}(\alpha_i))$.

7.2 The Functionality $\mathcal{F}_{\mathsf{triples}}$

On a high level, the functionality non-robustly generates a number of triples, and, in case of failure, identifies a set containing a significant number of actively- or fail-corrupted parties.

The parties input to $\mathcal{F}_{\mathsf{triples}}$ the desired number of triples ℓ and the configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$. Then, the adversary decides on one of three outcomes: (1) The detectable computation succeeds, and ℓ triples shared with degree $d = t_a + t_p$ among the parties in $\mathcal{P}'$ are generated; (2) The adversary disrupted the protocol and a set containing an active party is identified and outputted to all parties; (3) The adversary disrupted the protocol and a set of fail-corrupted parties is outputted. In the latter two cases, the adversary chooses the outputted set, but a sufficient fraction of parties in the set must be actually corrupted (if this is not satisfied, the parties receive the shared triples).

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

The functionality receives sets of corrupted parties $\mathcal{P}_a, \mathcal{P}_p, \mathcal{P}_f$. Let $d = t_a + t_p$.

```
1: Receive from each party a number $\ell$ and a valid configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$.
2: If these values (ignoring $\perp$) are not consistent among parties, or if $(\mathcal{P}', t'_a, t'_p, t'_f)$ is not valid, execute
   Complete Break Down. Else, send $(\textsc{Ok}, \ell, \mathcal{P}', t'_a, t'_p, t'_f)$ to the adversary and proceed as follows.
3: Receive from the adversary a message M, which is processed as follows:
   – If $\mathrm{M} = (\textsc{Triples}, ((a_j^{(k)}, b_j^{(k)}, c_j^{(k)}))_{k=1\dots\ell, P_j\in(\mathcal{P}_a\cup\mathcal{P}_p)})$, then generate $\ell$ triples shared among the par-
     ties in $\mathcal{P}'$ as follows. For each triple $k$, choose random $a^{(k)}$ and $b^{(k)}$, and let $c^{(k)} = a^{(k)}b^{(k)}$. Then, for
     each $k = 1, \dots, \ell$ and $\mathsf{x} \in \{a, b, c\}$, choose the polynomial $g_{\mathsf{x}}^{(k)}$ at random from the set of all polynomi-
     als of degree at most $d$, going through the point $(0, \mathsf{x}^{(k)})$ and all points in $\{(\alpha_j, \mathsf{x}_j^{(k)}) \mid P_j \in (\mathcal{P}_a\cup\mathcal{P}_p)\}$.
     Send to each $P_i \in \mathcal{P}'$ its shares $(g_a^{(k)}(\alpha_i), g_b^{(k)}(\alpha_i), g_c^{(k)}(\alpha_i))$.
   – If $\mathrm{M} = (\textsc{ActiveSet}, E)$, where $E \subseteq \mathcal{P}'$ is a set such that $|E\cap\mathcal{P}_a| \geq |E|/2$, then send $(\textsc{ActiveSet}, E)$
     to the parties.
   – If $\mathrm{M} = (\textsc{CrashSet}, E)$, where $E \subseteq \mathcal{P}'$, $E \subseteq \mathcal{P}_f \cup \mathcal{P}_a$ and $E \neq \emptyset$, then send $(\textsc{CrashSet}, E)$ to the
     parties.
   – Any other message is treated as $(\textsc{Triples}, ((0,0,0))_{k=1\dots\ell, P_j\in(\mathcal{P}_a\cup\mathcal{P}_p)})$.
```

7.3 Realizing $\mathcal{F}_{\mathsf{prepPhase}}$ in the $\mathcal{F}_{\mathsf{triples}}$-hybrid Model

We now present the protocol $\mathsf{PreparationPhase}$ that realizes $\mathcal{F}_{\mathsf{prepPhase}}$ in the $\mathcal{F}_{\mathsf{triples}}$-hybrid model. The protocol divides the L into $t_a + t_f$ segments of length ℓ and sequentially calls $\mathcal{F}_{\mathsf{triples}}$ with input ℓ. It starts with $\mathcal{P}'' = \mathcal{P}'$ and in case of a disruption, eliminates parties from $\mathcal{P}''$. The outputs of all parties is the resulting set $\mathcal{P}''$ and, for parties in $\mathcal{P}''$, the shares of the triples.

Protocol $\mathsf{PreparationPhase}$ $(L, (\mathcal{P}', t'_a, t'_p, t'_f))$

```
Let $\ell = \lceil \frac{L}{t_a+t_f} \rceil$ and $d = t_a + t_p$.
Set $\mathcal{P}'' = \mathcal{P}'$, $t''_a = t'_a$, $t''_p = t'_p$ and $t''_f = t'_f$. For each segment $k = 1 \dots (t_a + t_f)$ do:
1: Send $\ell$ and $(\mathcal{P}'', t''_a, t''_p, t''_f)$ to $\mathcal{F}_{\mathsf{triples}}$.
2: If the output is $(\textsc{ActiveSet}, E)$, then set $\mathcal{P}'' = \mathcal{P}'' \setminus E$ and $t''_a = t''_a - |E|/2$, and repeat Step 1.
3: Else if the output is $(\textsc{CrashSet}, E)$, then set $\mathcal{P}'' = \mathcal{P}'' \setminus E$ and $t''_f = t''_f - |E|$, and repeat Step 1.
4: Else, store the sharings received from $\mathcal{F}_{\mathsf{triples}}$ and continue to the next segment.
Every $P_i$ outputs the configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$. Moreover, every $P_i \in \mathcal{P}''$ outputs the first $L$ triples of
shares received from $\mathcal{F}_{\mathsf{triples}}$.
```

Theorem 1. *Assuming that $3t_a + 2t_p + t_f < n$, the protocol* $\mathsf{PreparationPhase}$ *securely realizes* $\mathcal{F}_{\mathsf{prepPhase}}$ *in the* $\mathcal{F}_{\mathsf{triples}}$*-hybrid model, in the presence of a static mixed adversary.*

Proof. The simulator $\mathcal{S}_{\mathsf{prepPhase}}$ only needs to simulate the interaction with the ideal functionality $\mathcal{F}_{\mathsf{triples}}$. This is done by simply executing the code of $\mathcal{F}_{\mathsf{triples}}$.

Throughout the execution in the real world, we have the following invariant: $(\mathcal{P}'', t''_a, t''_p, t''_f)$ is valid and the shares stored by parties in $\mathcal{P}''$ form correct d-sharings of triples. The former follows from the observation that $n'' - 2t''_a - t''_f$ is preserved when parties are eliminated (this is trivially guaranteed by $\mathcal{F}_{\mathsf{triples}}$). This also implies that $3t''_a + 2t''_p + t''_f < n''$. For the latter, notice that a d-sharing in $\mathcal{P}'$ is also a correct d-sharing in $\mathcal{P}''$, since d is constant. The above invariant shows that the outputs are the same in the real and in the ideal world. [12] □

7.4 Realizing $\mathcal{F}_{\mathsf{triples}}$

We first construct two auxiliary protocols: The first protocol generates a number of double sharings of random values, using hyper-invertible matrices. This double-sharing protocol can then be used in the second protocol to detectably generate a number of multiplication triples. The triple-generation protocol, together with fault detection and fault localization, can be used to realize $\mathcal{F}_{\mathsf{triples}}$.

Generating double sharings. The goal of the protocol DoubleShareRandom is to detectably generate a number of (d, d')-sharings of uniformly random values, unknown to the adversary, assuming that $3t'_a + 2t'_p + t'_f < n'$. The degrees d and d' of the outputted sharings can be arbitrary.

We use the trivial protocol Share, which, on input a sharing degree d and a value s from a party P_i, generates a (possibly inconsistent) d-sharing of s.

Protocol Share$(P_i, d, s, (\mathcal{P}', t'_a, t'_p, t'_f))$

P_i chooses a random polynomial g of degree d and sends to every party $P_j \in \mathcal{P}'$ its share $s_j = g(\alpha_j)$.

The protocol generates the (d, d')-sharings in buckets of size at most $n' - 2t'_a - t'_p - \min(t'_a, t'_p)$. The idea is to, for each bucket, use Share to generate n' double-sharings of random values, each value chosen by a different party. Up to t'_a of these sharings might be inconsistent and up to $t'_a + t'_p$ of them might be known to the adversary. Then, we apply to this vector of sharings a (fixed) hyper-invertible matrix. First, this ensures that at least $n' - t_a - t_p$ of the resulting sharings contain uniformly random values, unknown to the adversary. Moreover, hyper-invertibility guarantees that if there exists a set of t'_a resulting sharings which are consistent, then all sharings are consistent. We exploit this fact by reconstructing $2t'_a$ of the resulting sharings, each towards a different party, who then checks the consistency of the sharing it received. If no correct party notices an inconsistency, then all sharings must be consistent. This verification step reveals to the adversary additional $\min(2t'_a, t'_a + t'_p)$ shared values, hence, only $n' - t_a - t_p - \min(2t'_a, t'_a + t'_p) = n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ remain private. To generate any number ℓ of double sharings, the procedure sketched above is invoked a number of times (in parallel).

Protocol DoubleShareRandom$(d, d', \ell, (\mathcal{P}', t'_a, t'_p, t'_f))$

The ℓ sharings are generated in buckets of size $n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ (with the last bucket possibly smaller). Generate each bucket of size l using the following procedure:

1: Every party $P_i \in \mathcal{P}'$ chooses s_i at random and double-shares it among $\mathcal{P}'$ by invoking Share$(P_i, d, s_i, (\mathcal{P}', t'_a, t'_p, t'_f))$ and Share$(P_i, d', s_i, (\mathcal{P}', t'_a, t'_p, t'_f))$.

2: The parties compute locally $([r_1]_{d,d'}, \ldots, [r_{n'}]_{d,d'})^T = M([s_1]_{d,d'}, \ldots, [s_{n'}]_{d,d'})^T$, using the shares of $s_1, \ldots, s_{n'}$, where M is a fixed hyper-invertible matrix of size $n' \times n'$.

3: For every $P_i \in \mathcal{P}'$ and $j \in \{1, \ldots, 2t'_a\}$, P_i sends its shares of $[r_j]_{d,d'}$ to P_j.

[12] Observe that we do not need that in case $\mathcal{F}_{\mathsf{triples}}$ sends $(\textsc{CrashSet}, E)$, the parties in E are actually not correct. It is enough that they are in $\mathcal{P}_f$.

```
4:  Every $P_j$ with $j \in \{1, \ldots, 2t'_a\}$ verifies that the values it got define a correct $(d, d')$-sharing, that is,
    that all shares of the $d$-sharing lie on a polynomial $g$ of degree $d$, that all shares of the $d'$-sharing
    lie on a polynomial $g'$ of degree $d'$, and that $g(0) = g'(0)$. If any of these conditions does not hold,
    $P_j$ gets unhappy.
5:  The $l$ sharings generated in this bucket are $[r_{n'-l+1}]_{d,d'}, \ldots, [r_{n'}]_{d,d'}$.
Output all sharings generated in all buckets (that is, $\ell$ sharings in total).
```

The communication cost of DoubleShareRandom can be seen to be $O(\ell n'\kappa + n'^2\kappa)$. Hence, for large enough ℓ, the amortized complexity of generating one double-sharing is $O(n'\kappa)$.

Lemma 3. *Assuming that $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid and the values inputted by the parties are consistent,* DoubleShareRandom *detectably generates ℓ correct (d, d')-sharings, where for each sharing, the shared value is uniformly random given the view of the adversary. The communication complexity of* DoubleShareRandom *is $O(\ell n'\kappa + n'^2\kappa)$.*

Proof. Consider one bucket, in which l (d, d')-sharings are generated, where $l \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$.

Correctness: Assume that no crash occurred and that after the protocol all correct parties are happy. All values s_i generated by the correct parties in Step 1 are double-shared correctly. Thus, at least $n' - t'_a$ sharings $[s_i]_{d,d'}$ are correct. Moreover, at least t'_a out of $2t'_a$ sharings $[r_i]_{d,d'}$ verified in Step 4 are verified by correct parties and, thus, must be correct (as otherwise a correct party would become unhappy). Together, this gives n' correct sharings. Since M is hyper-invertible, any other sharing can be written as an affine combination of these correct sharings. Any affine combination of correct (d, d')-sharings is also a correct (d, d')-sharing, hence all involved sharings are correct.

Secrecy: The values known to the adversary are at most: $t'_a + t'_p$ values s_i (those chosen by passively or actively corrupted parties) and $\min(2t'_a, t'_a + t'_p) = t'_a + \min(t'_a, t'_p)$ values r_i (those possibly revealed to such parties). With these $2t'_a + t'_p + \min(t'_a, t'_p)$ values fixed, there is a bijective mapping between the actual $l \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ values r_i whose sharings are generated by the protocol and any other l values s_i, generated by honest or only fail-stop corrupted parties. Therefore, the values contained in the generated sharings are uniform random and independent of the view of the adversary.

Completeness: If all parties behave according to the protocol, correctness of DoubleShareRandom is trivial and security follows from the above argument.

Complexity: In each bucket the parties communicate $O(n'^2\kappa)$ bits, which follows by inspection of the protocol. Moreover, each bucket (but the last one) contains $n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ double-sharings, which can be seen to be at least $\frac{1}{5}n'$ [13], hence, there are at most $5\ell/n' + 1$ buckets. Therefore, the overall communication complexity of DoubleShareRandom is $O((5\ell/n' + 1)n'^2\kappa) = O(\ell n'\kappa + n'^2\kappa)$. □

Generating multiplication triples. The goal of the protocol GenerateTriples is to detectably generate a number of random multiplication triples, shared in an arbitrary degree d, assuming that $3t'_a + 2t'_p + t'_f < n'$. The idea is that GenerateTriples invokes DoubleShareRandom, in order to generate random double-sharings $[a_i]_{d,d'}, [b_i]_{d,d'}$ and $[r_i]_{d,2d'}$, where $d' = t'_a + t'_p$. The parties then use the d'-sharings of a_i and b_i, and the $2d'$-sharings of r_i to locally compute $2d'$-sharings of the blinded products $e_i = a_i b_i - r_i$ as $[e_i]_{2d'} = [a_i]_{d'} [b_i]_{d'} - [r_i]_{2d'}$. These blinded products are then publicly reconstructed using RecPub and the d-sharings of the products c_i are computed as $[c_i]_d = [r_i]_d + e_i$, using the d-sharings of r_i.

[13] If we set $m = 3t'_a + 2t'_p$, then $t'_a \geq m/5$ or $t'_p \geq m/5$. It follows that $n' - 2t'_a - t'_p - \min(t'_a, t'_p) = n' - m + t'_a + t'_p - \min(t'_a, t'_p) = n' - m + \max(t'_a, t'_p) + \min(t'_a, t'_p) - \min(t'_a, t'_p) \geq n'/5$.

Protocol GenerateTriples$(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f))$

```
1: The parties invoke DoubleShareRandom$(d, d', 2\ell, (\mathcal{P}', t'_a, t'_p, t'_f))$ and
   DoubleShareRandom$(d, 2d', \ell, (\mathcal{P}', t'_a, t'_p, t'_f))$, where $d' = t'_a + t'_p$, to generate random double sharings
   $[a_1]_{d,d'}, \dots, [a_\ell]_{d,d'}$, $[b_1]_{d,d'}, \dots, [b_\ell]_{d,d'}$ and $[r_1]_{d,2d'}, \dots, [r_\ell]_{d,2d'}$.
2: For $i \in \{1, \dots, \ell\}$, the parties compute locally a $2d'$-sharing of $e_i = a_i b_i - r_i$ as $[c_i]_{2d'} = [a_i]_{d'} [b_i]_{d'} -$
   $[r_i]_{2d'}$.
3: The parties invoke RecPub$(2d', \ell, (\mathcal{P}', t'_a, t'_p, t'_f), [e_1]_{2d'}, \dots, [e_\ell]_{2d'})$.
4: For $i \in \{1, \dots, \ell\}$, the parties compute locally a $d$-sharing of $c_i = a_i b_i$ as $[c_i]_d = [r_i]_d + e_i$.
5: Output the $\ell$ triples $([a_1]_d, [b_1]_d, [c_1]_d), \dots, ([a_\ell]_d, [b_\ell]_d, [c_\ell]_d)$.
```

Lemma 4. *Assuming that $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid and the values inputted by the parties are consistent,* GenerateTriples *detectably generates ℓ triples of sharings, where each sharing is a correct d-sharing, and for each triple $([a]_d, [b]_d, [c]_d)$, the shared values a and b are uniformly random given the view of the adversary, and $c = ab$. The communication complexity of* GenerateTriples *is $O(\ell n' \kappa + n'^2 \kappa)$.*

Proof. Correctness and Secrecy: Since $3t'_a + 2t'_p + t'_f < n'$, we get that the degree $2d'$ of the sharings used in Step 3 fulfills $2d' = 2t'_a + 2t'_p < n' - t'_a$. Hence, correctness and secrecy of outputted triples follows from Lemmas 8 and 3.

Complexity: The claim trivially follows by inspection of the protocol and Lemmas 8 and 3. □

The protocol. The following protocol Triples realizes $\mathcal{F}_{\mathsf{triples}}$.

Protocol Triples $(\ell, (\mathcal{P}', t'_a, t'_p, t'_f))$

```
1: Every party in $\mathcal{P}'$ sets its happy-bit to "happy".
   // Detectable Computation
2: The parties invoke GenerateTriples$(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f))$ for $d = t_a + t_p$.
   // Fault Detection
3: The parties invoke FaultDetect. If, as a result, they are all happy, they output the generated triples.
   Otherwise, they do the following.
   // Fault Localization
4: Let $P_r$ be the party with the smallest index in $\mathcal{P}'$. Every $P_i \in \mathcal{P}'$ sends to $P_r$ the randomness $R_i$ it
   used in the two previous steps and the messages $M_i$ it received in those steps. If $P_r$ does not receive
   values from some parties, it uses instead the default values and proceeds.[14]
5: For each $P_i$, $P_r$ reproduces all messages that $P_i$ should have sent, using $R_i$ and $M_i$, and, for each $P_j$,
   compares them with the messages $P_j$ claims to have received from $P_i$ (as specified in $M_j$). Then, $P_r$
   broadcasts a tuple $(l, P_i, P_j, x, x')$, such that the $l^{th}$ message $P_j$ claims to have received from $P_i$ was
   $x'$, while according to $P_i$ it should have been $x$.
6: $P_i$ broadcasts whether it agrees with $P_r$ (i.e., whether the $l^{th}$ message it sent to $P_j$ was $x$). $P_j$
   broadcasts whether it agrees with $P_r$ (i.e., whether if the $l^{th}$ message it received from $P_i$ was $x'$).
7: If $P_i$ disagrees, every party sets $E = \{P_i, P_r\}$. If $P_j$ disagrees, every party sets $E = \{P_j, P_r\}$. Otherwise,
   every party sets $E = \{P_i, P_j\}$.
8: For every $P_h \in \{P_i, P_j, P_r\}$, the parties in $\mathcal{P}'$ invoke Heartbeat$(P_h)$.
9: If the output of every invocation of Heartbeat is "alive", output $E$. Otherwise, output the set of parties,
   for whom the output of Heartbeat was "crashed".
```

Theorem 2. *Assuming that $3t_a + 2t_p + t_f < n$, the protocol* Triples *securely evaluates $\mathcal{F}_{\mathsf{triples}}$ in the presence of a static mixed adversary.*

[14] If P_r does not get the values from some party P_i, it could, instead of ignoring it, broadcast the index of such party. However, such solution would make the description of the protocol more involved.

Proof. We present the simulator $\mathcal{S}_{\mathsf{triples}}$. Roughly, since there are no private inputs to the protocol (only private outputs), $\mathcal{S}_{\mathsf{triples}}$ simulates the execution towards the adversary $\mathcal{A}$ by executing the protocol. The key point is to make sure that the outputs are distributed correctly.

Simulator $\mathcal{S}_{\mathsf{triples}}$

The simulator has black-box access to the adversary $\mathcal{A}$. It outputs whatever $\mathcal{A}$ outputs.

```
1: Receive $(\textsc{Ok}, \ell, \mathcal{P}', t'_a, t'_p, t'_f)$ from $\mathcal{F}_{\mathsf{triples}}$.
2: Execute $\mathsf{GenerateTriples}(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f))$ and $\mathsf{FaultDetect}$ on behalf of the correct parties in $\mathcal{P}' \setminus \mathcal{P}_a$:
   in each round, send the messages generated by the protocol to $\mathcal{A}$, receive the messages from corrupted
   parties and information about crashes, and compute the next messages accordingly.
3: If the parties do not agree on the output of $\mathsf{FaultDetect}$, abort.
4: Otherwise, if this output is "happy", do as follows: For each triple $k = 1 \dots \ell$, compute the shares of the
   corrupted parties in $\mathcal{P}'$ that would result from the protocol (this is fully determined by the exchanged
   messages) and choose random shares for the corrupted parties not in $\mathcal{P}'$. Send these shares, together
   with the command $\textsc{Triples}$, to $\mathcal{F}_{\mathsf{triples}}$.
5: Otherwise (i.e., the output is "unhappy"), continue by executing fault localization. If the correct parties
   do not agree on the resulting set $E$, abort. Otherwise, send to $\mathcal{F}_{\mathsf{triples}}$ $(\textsc{ActiveSet}, E)$ or $(\textsc{CrashSet}, E)$,
   depending on the result of fault localization.
```

Assume that the honest parties input consistent values ℓ, $\mathcal{P}'$, t'_a, t'_p and t'_f, and that the set $\mathcal{P}'$ is valid (otherwise, the simulation is trivial). This means that the preconditions for Lemmas 1, 2 and 4 are met. By consistency of $\mathsf{FaultDetect}$ (Lemma 1), $\mathcal{S}_{\mathsf{triples}}$ does not abort in Step 3. Then, $\mathcal{S}_{\mathsf{triples}}$ sends the command $\textsc{Triples}$ if and only if the result of $\mathsf{FaultDetect}$ is "happy", which is the same as in the real world. Hence, we can consider two cases: (1) the simulator sends $\textsc{Triples}$, and (2) he sends $\textsc{ActiveSet}$ or $\textsc{CrashSet}$.

Case $\textsc{Triples}$**.** Consider any triple and the corresponding sharing polynomials g_a, g_b and g_c. In the real world, by correctness of $\mathsf{FaultDetect}$, no party is crashed and no correct party is unhappy at the end of $\mathsf{GenerateTriples}$. Therefore, the secrecy stated in Lemma 4 guarantees that the free coefficients a and b of g_a and g_b are uniformly random and independent of the view of the adversary. Moreover, correctness stated in Lemma 4 guarantees that the free coefficient of g_c is $c = ab$. Therefore, the free coefficients are distributed identically as in the ideal world.

By correctness (Lemma 4), the degree of all polynomials is at most d. It is easy to see that in both worlds the polynomials are uniformly random among those consistent with the shares that would be outputted by the corrupted parties and with the free coefficients distributed as above.

Case $\textsc{ActiveSet}$ or $\textsc{CrashSet}$**.** In this case it is enough to argue correctness (note that here the parties have no secret inputs and the ideal functionality is deterministic). The set E sent by $\mathcal{S}_{\mathsf{triples}}$ is the same as in the protocol.

The simulator sends $\textsc{CrashSet}$ if and only if the output of $\mathsf{Heartbeat}$ for some parties in $\mathcal{P}'$ was not "alive". In this case, E contains no correct party, by completeness of $\mathsf{Heartbeat}$ (Lemma 2).

Now consider the other case, where $\mathcal{S}_{\mathsf{triples}}$ sends $\textsc{ActiveSet}$. By correctness of $\mathsf{Heartbeat}$, none of P_i, P_j and P_k was crashed at the end of Step 7. It follows that if in Step 4 P_r received no messages from P_i (or P_j), then P_i (or P_j) is malicious and could have sent the default values taken by P_r anyway. Consider how the set E is chosen in Step 7. If P_i (or P_j) disagrees, then clearly it makes a conflicting claim with P_r. On the other hand, if none of P_i and P_j disagrees with P_r, then one of P_i and P_j must be malicious, because they disagree on the message sent from P_i to P_j. Hence, E contains two parties making conflicting claims. One of these parties must be actively corrupted. □

7.5 Complexity

Consider first the complexity of the protocol $\mathsf{Triples}$. By Lemmas 4 and 1, executing $\mathsf{GenerateTriples}$ and $\mathsf{FaultDetect}$ requires communicating $O(\ell n\kappa + n^2\kappa + \mathcal{BA}(1))$ bits. For the complexity of fault localization, observe that the total number of bits needed to send the messages M_i is exactly the communication complexity of GenerateTriples and FaultDetect, and that, asymptotically, sending the values R_i does not add to this complexity. Together with broadcasts and Heartbeat, this results in $O(\ell n\kappa + n^2\kappa + \mathcal{BA}(\kappa))$ bits for the whole protocol Triples.

For the final protocol PreparationPhase, notice that the adversary can make the parties repeat a segment at most $t_a + t_f$ times. This means that Triples is executed at most $2(t_a + t_f)$ times. Hence, the protcol PreparationPhase communicates PreparationPhase is $O(Ln\kappa + (t_a + t_f)(n^2\kappa + \mathcal{BA}(\kappa)))$, which amounts to $O(Ln\kappa + n^3\kappa)$.

8 Evaluating Any Circuit

The goal is to realize the functionality $\mathcal{F}_{\mathsf{mpc}}$, that evaluates a circuit inputted by the parties. This is done in two phases: preprocessing and evaluation. The previous section explained how to execute preprocessing, as formalized by $\mathcal{F}_{\mathsf{prepPhase}}$. In this section, we explain how to complete the circuit evaluation by executing the evaluation phase.

Towards this goal, we first define a multiplication functionality $\mathcal{F}_{\mathsf{mult}}$, which can be used to evaluate a number of multiplication gates on the same depth. We then realize $\mathcal{F}_{\mathsf{mpc}}$ in the $(\mathcal{F}_{\mathsf{prepPhase}}, \mathcal{F}_{\mathsf{mult}})$-hybrid model. Then, we proceed to realize $\mathcal{F}_{\mathsf{mult}}$ in the $\mathcal{F}_{\mathsf{rec}}$-hybrid model, where $\mathcal{F}_{\mathsf{rec}}$ robustly publicly reconstructs a number of sharings. Finally, we realize $\mathcal{F}_{\mathsf{rec}}$ with the help of the detectable protocol RecPub and player elimination.

8.1 The Functionality $\mathcal{F}_{\mathsf{mpc}}$

The following functionality $\mathcal{F}_{\mathsf{mpc}}$ evaluates any circuit inputted by the parties. Recall that, since the adversary can always prevent fail-corrupted parties from giving input, the functionality is defined on an extended input domain, where a party can input $\perp$, meaning that it gives no input. $\mathcal{F}_{\mathsf{mpc}}$ deals with input $\perp$ by evaluating all input gates of a given perty using the default input 0. Moreover, $\mathcal{F}_{\mathsf{mpc}}$ sends to all parties the set of all parties who inputted $\perp$. This way, parties know what circuit was actually evaluated.

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

1: Receive from each party a circuit C with a topological order on the gates and the party's inputs to C. (Execute Complete Break Down if different values C are received, or if C cannot be evaluated.)
2: For each party whose input was $\perp$, set all inputs to C to 0.
3: Evaluate C and send outputs to the corresponding parties. Send to all parties the set of parties who inputted $\perp$.

8.2 The Functionality $\mathcal{F}_{\mathsf{mult}}$

Given two vectors of sharings and a vector of multiplication triples, the functionality $\mathcal{F}_{\mathsf{mult}}$ outputs a vector of shared products.

In more detail, the inputs to $\mathcal{F}_{\mathsf{mult}}$ include the current configuration and the number L of multiplications gates to evaluate. The parties in the current party set additionally input the shares of gate inputs. Since our protocol uses circuit randomization, they also input the shares of multiplication triples.

Recall that with circuit randomization, we reveal to the adversary the differences between inputs and values in the triples. Accordingly, $\mathcal{F}_{\mathsf{mult}}$ reveals this as well. Observe that implicitly this means that the multiplication preserves the privacy of inputs as long as the value in the triple is unknown to the adversary. Since our protocol employs player elimination, $\mathcal{F}_{\mathsf{mult}}$ also outputs a reduced valid configuration, chosen by the adversary.

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

The functionality receives sets of corrupted parties $\mathcal{P}_a, \mathcal{P}_p, \mathcal{P}_f$. Let $d = t_a + t_p$.

```
1: Receive from the valid parties a value $L$ and a configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$. Moreover, from each valid
   $P_i \in \mathcal{P}'$ receive shares $(x_i^{(k)}, y_i^{(k)}, a_i^{(k)}, b_i^{(k)}, c_i^{(k)})_{k=1\ldots L}$, corresponding to the inputs $x^{(k)}, y^{(k)}$ to the gates
   and the multiplication triples $(a^{(k)}, b^{(k)}, c^{(k)})$.
2: Execute Complete Break Down if either:
   – $L$, $(\mathcal{P}', t'_a, t'_p, t'_f)$ are not consistent, or $(\mathcal{P}', t'_a, t'_p, t'_f)$ is not valid, or
   – For any $k$, the received shares $x_i^{(k)}$ and $y_i^{(k)}$ do not form a correct $d$-sharing or the shares $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}$
     do not form a correct triple.
3: For each $k = 1 \ldots L$ and $\mathsf{x} \in \{x, y, a, b, c\}$, compute the polynomial $g_{\mathsf{x}}^{(k)}$ of degree at most $d$ such that
   all points in $\{(\alpha_i, \mathsf{x}_i^{(k)}) \mid P_i \in \mathcal{P}'\}$ lie on $g_{\mathsf{x}}^{(k)}$. Let $g_r^{(k)} = g_x^{(k)} - g_a^{(k)}$ and $g_s^{(k)} = g_y^{(k)} - g_b^{(k)}$.
4: Send to the adversary the values
   $(\textsc{Ok}, L, \mathcal{P}', t'_a, t'_p, t'_f, (g_r^{(k)}, g_s^{(k)})_{k=1\ldots L}, ((g_a^{(k)}(\alpha_j), g_b^{(k)}(\alpha_j), g_c^{(k)}(\alpha_j)))_{k=1\ldots L, P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)})$.
5: Receive from the adversary:
   – A valid configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ with $\mathcal{P}'' \subseteq \mathcal{P}'$,
   – For each $P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)$, shares $(z_j^{(k)})_{k=1\ldots L}$.
   (Set $\mathcal{P}'' = \mathcal{P}'$, $t''_{\mathsf{x}} = t'_{\mathsf{x}}$ and $z_j^{(k)} = 0$ if no valid values are received.)
6: Send $(\mathcal{P}'', t''_a, t''_p, t''_f)$ to all parties. Moreover, for each $k = 1 \ldots L$, choose the polynomial $g_z^{(k)}$ at random
   from the set of all polynomials of degree at most $d$, going through all points in $\{(\alpha_j, z_j^{(k)}) \mid P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)\}$
   and the point $(0, g_x^{(k)}(0) \cdot g_y^{(k)}(0))$. Send to each $P_i \in \mathcal{P}''$ its shares $g_z^{(k)}(\alpha_i)$.
```

8.3 Realizing $\mathcal{F}_{\mathsf{mpc}}$ in the $(\mathcal{F}_{\mathsf{prepPhase}}, \mathcal{F}_{\mathsf{mult}})$-hybrid Model

The following protocol realizes $\mathcal{F}_{\mathsf{mpc}}$ in the $(\mathcal{F}_{\mathsf{prepPhase}}, \mathcal{F}_{\mathsf{mult}})$-hybrid model. All parties input the same circuit C, and every party inputs its inputs to the gates. Then, the protocol proceeds in four phases: First, the parties call $\mathcal{F}_{\mathsf{prepPhase}}$ to generate $c_i + c_m + c_r$ triples, needed for input, multiplication and random gates. Second, the parties evaluate all input gates, using circuit randomization. Then, in the computation phase, the affine gates are evaluated locally, and multiplication gates are evaluated with the help of $\mathcal{F}_{\mathsf{mult}}$. Finally, all output gates are evaluated by invoking the robust reconstruction protocol RecPrivRobust.

Inputs. We first present a subroutine InputGate that evaluates one input gate. We do not prove any guarantees about this protocol, but we still present it separately for clarity.

Intuitively, InputGate implements the randomization technique, where parties input a sharing of a random value r among the parties in $\mathcal{P}'$. This value is then reconstructed towards P_i, who broadcasts his blinded input $s - r$. Recall that in the mixed setting, the adversary can prevent a fail-corrupted P_i from giving input. Accordingly, InputGate returns "success" or "false", indicating whether the input was successfully given. However, we still require that the adversary cannot modify inputs of a fail-corrupted P_i. To achieve this, we additionally execute on him Heartbeat.

Protocol InputGate$(P_i, [r]_d, (\mathcal{P}', t'_a, t'_p, t'_f))$

The protocol evaluates the input gate, where $P_i \in \mathcal{P}$ inputs s.

```
1: The parties invoke RecPrivRobust$(P_i, d, (\mathcal{P}', t'_a, t'_p, t'_f), [r]_d)$.
2: The parties invoke broadcast, where $P_i$ sends his blinded input $e = s - r$.
3: The parties invoke Heartbeat$(P_i)$.
4: If the output is "alive", then each $P_j \in \mathcal{P}$ outputs "success". Additionally, each $P_j \in \mathcal{P}'$ outputs the
   share of the output wire $e + r_j$, where $r_j$ is $P_j$'s share of $r$. Otherwise, each $P_j \in \mathcal{P}$ outputs "fail".
```

The MPC protocol. We now present the overall protocol. Throughout the execution, all parties in $\mathcal{P}$ keep track of the current configuration and of the set D of parties who failed to give their inputs. Moreover, the parties in the current set $\mathcal{P}'$ keep shares of the values on the wires computed so far.

Protocol $\mathsf{MPC}(C)$

INITIALIZATION:

```
1: Each party sets $\mathcal{P}' = \mathcal{P}$ and $t'_{\mathsf{x}} = t_{\mathsf{x}}$ for all $\mathsf{x} \in \{a, b, c\}$. Moreover, it initializes the set of parties who do not give input $D = \emptyset$.
2: Each $P_i \in \mathcal{P}$ sends $(c_i + c_m + c_r, (\mathcal{P}, t_a, t_p, t_f))$ to $\mathcal{F}_{\mathsf{prepPhase}}$. Then, each $P_i \in \mathcal{P}$ receives and stores the output $(\mathcal{P}', t'_a, t'_p, t'_f)$. The parties in $\mathcal{P}'$ receive and store the shares of triples.
```

INPUT: Evaluate the input gates one by one.

```
1: For each input gate, where $P_i \in \mathcal{P}$ inputs $s$, proceed as follows. Let $([a]_d, [b]_d, [c]_d)$ denote the associated triple. Invoke $\mathsf{InputGate}(P_i, [a]_d, (\mathcal{P}', t'_a, t'_p, t'_f))$. If the output is "success", then every $P_j \in \mathcal{P}'$ stores the share of the output wire, outputted by InputGate. Otherwise, every $P_j \in \mathcal{P}$ adds $P_i$ to $D$.
2: For each $P_i \in D$, every $P_j \in \mathcal{P}'$ sets the share of all input gates where $P_i$ gives input to 0.
```

COMPUTATION: Evaluate the circuit in the topological order as follows.

```
– To evaluate an affine gate that computes the function $f$, do:
    1: Each $P_i \in \mathcal{P}'$ computes the share of the output wire as $f(x_i^{(1)}, \dots, x_i^{(l)})$, where $x_i^{(1)}, \dots, x_i^{(l)}$ denote its shares of input wires.
– To evaluate $L$ multiplication gates, do:
    1: For $k = 1 \dots L$, let $(\hat{a}_i^{(k)}, \hat{b}_i^{(k)}, \hat{c}_i^{(k)})$ and $\hat{x}_i^{(k)}, \hat{y}_i^{(k)}$ denote the shares of $P_i$ of to the associated triples and the gate inputs, respectively.
    2: Each $P_i \in \mathcal{P}$ sends $(L, (\mathcal{P}', t'_a, t'_p, t'_f))$ to $\mathcal{F}_{\mathsf{mult}}$. Each $P_i \in \mathcal{P}'$ sends in addition $(\hat{x}_i^{(k)}, \hat{y}_i^{(k)}, \hat{a}_i^{(k)}, \hat{b}_i^{(k)}, \hat{c}_i^{(k)})_{k=1\dots,L}$. Then, each $P_i \in \mathcal{P}$ receives from $\mathcal{F}_{\mathsf{mult}}$ the configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$, and sets $\mathcal{P}' = \mathcal{P}''$ and $t'_{\mathsf{x}} = t''_{\mathsf{x}}$ for $\mathsf{x} \in \{a, p, f\}$. Moreover, every $P_i \in \mathcal{P}''$ receives from $\mathcal{F}_{\mathsf{mult}}$ the shares of the output wires $\hat{z}_i^{(k)}$.
```

OUTPUT: Every party outputs the set D. Moreover, the output gates are evaluated one by one. For each output gate, where $P_i \in \mathcal{P}$ receives a value, proceed as follows.

```
1: The parties invoke $\mathsf{RecPrivRobust}(P_i, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$, where $s$ denotes the value on the input wire to the gate.
```

Theorem 3. *Assuming that $3t_a + 2t_p + t_f < n$, the protocol* MPC *securely realizes any functionality in the $(\mathcal{F}_{\mathsf{prepPhase}}, \mathcal{F}_{\mathsf{mult}})$-hybrid model, in the presence of a static mixed adversary.*

Proof. On a high level, the simulator follows the evaluation of the circuit in the topological order, keeping track of the current configuration, the set D of gates with default input 0, and the passively and actively corrupted parties' shares of the values on the wires. We will refer to the latter as *corrupted shares.*

The key point in the simulation is to come up with blinded inputs to the input gates and to the multiplication gates (recall that $\mathcal{F}_{\mathsf{mult}}$ reveals these blinded inputs to the adversary). Recall that these values are blinded by the values a and b from the triples generated by the $\mathcal{F}_{\mathsf{prepPhase}}$. Since a and b are random and never revealed by the ideal functionality, the simulator $\mathcal{S}_{\mathsf{mpc}}$ can simply replace the blinded inputs by random values (the same way one would simulate one-time-pad encryption).

Note also that for $\mathcal{F}_{\mathsf{mult}}$, the simulator has to reveal the whole sharing polynomials. These polynomials can be interpolated given the corrupted shares of a and b, received from the adversary when emulating $\mathcal{F}_{\mathsf{prepPhase}}$ (if the there are too few corrupted parties to uniquely define the polynomial, $\mathcal{S}_{\mathsf{mpc}}$ chooses the polynomial at random).

Simulator $\mathcal{S}_{\mathsf{mpc}}$

The simulator has black-box access to the adversary $\mathcal{A}$. It outputs whatever $\mathcal{A}$ outputs.

INITIALIZATION: Emulate $\mathcal{F}_{\mathsf{prepPhase}}$.

```
1: Send $(\text{OK}, c_i + c_m + c_r, \mathcal{P}, t_a, t_p, t_f)$ to $\mathcal{A}$.
```

```
2: Receive and store $\mathcal{P}', t'_a, t'_p, t'_f$ and the shares $((a_j^{(k)}, b_j^{(k)}, c_j^{(k)}))_{k=1\dots L, P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)}$. Set $D = \emptyset$.

INPUT: For each input gate, where $P_i \in \mathcal{P}$ inputs $s$, do as follows.
1: Let $(a_j, b_j, c_j)_{P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)}$ be the corrupted shares corresponding to the associated triple.
2: if $P_i$ is passively or actively corrupted then
3:     Choose the polynomial $g_a$ at random from the set of all polynomials of degree at most $d$, going through all points in $\{(\alpha_j, a_j) \mid P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)\}$ (this defines a random shared value $a'$.) Using this, emulate InputGate towards $\mathcal{A}$. This can be done, since $\mathcal{S}_{\mathsf{mpc}}$ knows all inputs to RecPrivRobust, Heartbeat and broadcast (in the latter the only input is that of the corrupted $P_i$).
4: else
5:     Emulate InputGate, using random $e$ as $P_i$'s input to broadcast (note that only $P_i$ receives messages during RecPrivRobust).
6: end if
7: Compute and store the corrupted shares of the output wire of this gate as follows. If the result of InputGate is "success", then these shares are $a_j + e'$, where $e'$ denotes the output of broadcast. Else, the shares are 0.
8: If $P_i$ is fail-corrupted and the output of InputGate is "fail", add $P_i$ to $D$.
9: If $P_i$ is actively corrupted, then compute and store the input effectively used in the evaluation — $e' + g_a(0)$ (where $e'$ denotes the output of broadcast and $g_a$ is the polynomial from Step 3) if InputGate outputted "success" and $\perp$ otherwise.
Finally, for each gate where a $P_i \in D$ gives input, set the corrupted shares of the output wires to 0.

COMPUTATION:

– For an affine gate that computes the function $f$, do:
    1: Let $(x_j^{(1)}, \dots, x_j^{(l)})_{P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)}$ denote the corrupted shares of the input wires. The shares of the output wire are $(f(x_j^{(1)}, \dots, x_j^{(l)}))_{P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)}$.
– To emulate $\mathcal{F}_{\mathsf{mult}}$, do:
    1: For $k = 1 \dots L$ and $P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)$, let $(a_j^{(k)}, b_j^{(k)}, c_j^{(k)})$ and $x_j^{(k)}, y_j^{(k)}$ denote the corrupted shares of the associated triples and the gate inputs, respectively.
    2: For each $k = 1 \dots L$, choose random $r^{(k)}$ and $s^{(k)}$ and compute $r_j^{(k)} = x_j^{(k)} - a_j^{(k)} + r^{(k)}$ and $s_j^{(k)} = y_j^{(k)} - b_j^{(k)} + s^{(k)}$. Then, for each $k$ and $\mathsf{x} \in \{r, s\}$, choose $g_{\mathsf{x}}^{(k)}$ at random from the set of all polynomials of degree at most $d$, going through all points in $\{(\alpha_j, \mathsf{x}_j) \mid P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)\}$ and the point $(0, \mathsf{x}^{(k)})$.
    3: Send to the adversary $(\text{OK}, L, \mathcal{P}', t'_a, t'_p, t'_f, (g_r^{(k)}, g_s^{(k)})_{k=1\dots L}, ((a_j^{(k)}, b_j^{(k)}, c_j^{(k)}))_{k=1\dots L, P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)})$.
    4: Receive from the adversary the configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ and the shares $z_j^{(k)}$.
    5: Update $\mathcal{P}' = \mathcal{P}''$ and $t'_{\mathsf{x}} = t''_{\mathsf{x}}$ (or default if no valid values are received). Store $z_j^{(k)}$ as the corrupted shares of the output wires.

OUTPUT:
1: Specify the stored inputs effectively used by actively-corrupted parties, and the set $D$ of fail-corrupted parties whose input is replaced by $\perp$, and receive the outputs of actively- and passively-corrupted parties.
2: For every output gate where $P_i$ receives a value, do as follows. If $P_i$ is not passively- or actively-corrupted, no simulation is necessary. Otherwise, let $(x_j)_{P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)}$ denote the corrupted shares of the input wire of this gate, and let $s$ denote the output value, received from $\mathcal{F}_{\mathsf{mpc}}$. Choose $g_x$ at random from the set of all polynomials of degree at most $d$, going through all points in $\{(\alpha_j, x_j) \mid P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)\}$ and the point $(0, s^{(k)})$. Using $g_x$, compute the values inputted by the parties to RecPrivRobust and emulate the protocol towards $\mathcal{A}$.
```

We ignore the case of Complete Break Down (since then the simulation is trivial). Observe first that the current configuration is always valid both in the real and in the ideal world, which easily follows by inspection (recall that all functionalities execute Complete Break Down if this is not the case). Hence, in the following, we implicitly assume a valid configuration.

Phase INITIALIZATION**.** Here the simulation is trivial.

Phase Input**.** We show that the interaction with $\mathcal{A}$ is the same as in the real world. Since emulating Heartbeat is trivial given that $\mathcal{S}_{\mathsf{mpc}}$ knows when parties crash, we focus on the broadcast and private reconstruction. In case P_i is neither passively nor actively corrupted, then in RecPrivRobust there are no messages sent to corrupted parties, and the value a (randomly chosen by $\mathcal{F}_{\mathsf{prepPhase}}$ and not revealed to the adversary) perfectly blinds the input to broadcast. Otherwise, $\mathcal{S}_{\mathsf{mpc}}$ uses a random input to RecPrivRobust, which is clearly the same as in the real world. For the broadcast, observe that only the corrupted sender has input. So, $\mathcal{S}_{\mathsf{mpc}}$ can easily compute the messages sent by all other parties. P_i's messages are computed using his input, if he is passively corrupted, or by the adversary.

Furthermore, it is easy to see that the corrupted shares of the output wires used by the simulator are the same as in the real world.

Observe also that the set D and the inputs of corrupted parties are computed as in the real world (this is needed later, when we argue that the outputs are the same in both worlds).

Phase Computation**.** The corrupted shares maintained by $\mathcal{S}_{\mathsf{mpc}}$ throughout the simulation are the same as in the real world. This is clear for the input gates and the linear gates. It also trivially holds for multiplication gates, where these shares are in both cases chosen by $\mathcal{A}$.

It is left to show that $\mathcal{S}_{\mathsf{mpc}}$ perfectly emulates calls to $\mathcal{F}_{\mathsf{mult}}$. Consider one such call. Notice that the gates in the bunch are processed independently, so we focus on one gate with inputs x, y and the associated triple (a, b, c) (note that this triple is fresh and only used here).

In the real world, the parties input to $\mathcal{F}_{\mathsf{mult}}$ shares of correct triples (this is guaranteed by $\mathcal{F}_{\mathsf{prepPhase}}$). Hence, the parties receive from $\mathcal{F}_{\mathsf{mult}}$ the shares of $z = xy$, where the sharing polynomial is chosen to be consistent with the corrupted shares provided by $\mathcal{A}$. Moreover, the adversary receives (1) the whole sharing polynomials g_r, g_s, corresponding to the blinded products, and (2) the corrupted shares a_j, b_j, c_j. The free coefficients of the polynomials (1) are $r = x - a$ and $s = y - b$. Since a and b have been chosen by $\mathcal{F}_{\mathsf{prepPhase}}$ at random, the values r and s are also random and independent of the view of the adversary.

In the ideal world, the implicit value on the output wire for this gate is $z = xy$ as well. The sharing polynomial is also chosen by $\mathcal{S}_{\mathsf{mpc}}$ to be consistent with the corrupted shares provided by $\mathcal{A}$. Moreover, the corrupted shares (2) are given to $\mathcal{S}_{\mathsf{mpc}}$ by $\mathcal{F}_{\mathsf{prepPhase}}$, so their distribution is the same as in the real world. The same holds for the polynomials (1), since their free coefficients are in both cases random, and $\mathcal{S}_{\mathsf{mpc}}$ programs the rest of the polynomials to match the real world.

Phase Output**.** Observe that (1) the inputs sent by $\mathcal{S}_{\mathsf{mpc}}$ to $\mathcal{F}_{\mathsf{mpc}}$ are the same as the effective inputs used in the input gates (as showed when analyzing the Input phase), and (2) the circuit is computed correctly in the real world (which follows by inspection). It follows that the free coefficients of the polynomials corresponding to the sharings reconstructed by RecPrivRobust are the same in the real and in the ideal world. Moreover, in both cases these polynomials are random among all polynomials of degree at most d with the correct free coefficient, which are consistent with the corrupted shares known to $\mathcal{A}$. Hence, the inputs to RecPrivRobust are distributed identically in both worlds and the simulation is perfect.

□

8.4 The Functionality $\mathcal{F}_{\mathsf{rec}}$

The functionality $\mathcal{F}_{\mathsf{rec}}$ robustly reconstructs a number of values shared with degree $d = t_a + t_p$ among the parties in $\mathcal{P}'$. Our protocol employs player elimination, so the values are reconstructed towards all parties in a reduced set $\mathcal{P}''$. In particular, all parties output the resulting configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$, and all parties in $\mathcal{P}''$ output the reconstructed values. We do not consider privacy and, accordingly, all inputs are always revealed to the adversary. (Note that in our protocol, the adversary indeed may learn the whole sharing polynomials.)

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

```
The functionality receives sets of corrupted parties $\mathcal{P}_a, \mathcal{P}_p, \mathcal{P}_f$. Let $d = t_a + t_p$.
1: Receive from each party $L$ and a valid configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$. Moreover, from every valid $P_i \in \mathcal{P}'$
   receive $L$ shares $(s_i^{(k)})_{k=1...L}$.
2: Execute Complete Break Down if the received values $L$ and $(\mathcal{P}', t'_a, t'_p, t'_f)$ are not consistent among
   parties, or if $(\mathcal{P}', t'_a, t'_p, t'_f)$ is not valid, or if for any $k$ the received values $s_i^{(k)}$ do not form a correct
   $d$-sharing. Otherwise, proceed as follows.
3: For each sharing $k$, compute the polynomial $g^{(k)}$ of degree at most $d$ such that all points $(\alpha_i, s_i^{(k)})$
   received from parties in $\mathcal{P}'$ lie on $g^{(k)}$.
4: Send to the adversary $(\textsc{Ok}, L, \mathcal{P}', t'_a, t'_p, t'_f, (g^{(k)})_{k=1...L})$.
5: Receive from the adversary a valid configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ with $\mathcal{P}'' \subseteq \mathcal{P}'$. (Set $\mathcal{P}'' = \mathcal{P}'$ and $t''_{\mathsf{x}} = t'_{\mathsf{x}}$
   if no valid values are received.)
6: Send $(\mathcal{P}'', t''_a, t''_p, t'_f)'$ to all parties. Moreover, send to each $P_i \in \mathcal{P}''$ reconstructed values $(g^{(k)}(0))_{k=1...L}$.
```

8.5 Realizing $\mathcal{F}_{\mathsf{mult}}$ in the $\mathcal{F}_{\mathsf{rec}}$-hybrid Model

The following protocol Multiply realizes $\mathcal{F}_{\mathsf{mult}}$ in the $\mathcal{F}_{\mathsf{rec}}$-hybrid model, with the help of circuit randomization.

Protocol Multiply

```
// The input of each $P_i$ is $L$ and a valid configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$. Additionally, the input of each $P_i \in \mathcal{P}'$
includes $L$ shares $(x_i^{(k)}, y_i^{(k)}, a_i^{(k)}, b_i^{(k)}, c_i^{(k)})_{k=1...L}$.
1: Each $P_i$ sends $2L$ and $(\mathcal{P}', t'_a, t'_p, t'_f)$ to $\mathcal{F}_{\mathsf{rec}}$. Moreover, every $P_i \in \mathcal{P}'$ sends to $\mathcal{F}_{\mathsf{rec}}$ $2L$ shares $(r_i^{(k)})_{k=1...\ell}$
   and $(s_i^{(k)})_{k=1...\ell}$, where $r_i^{(k)} = x_i^{(k)} - a_i^{(k)}$ and $s_i{}^{(k)} = y_i{}^{(k)} - b_i{}^{(k)}$.
2: Each $P_i$ outputs $(\mathcal{P}'', t''_a, t''_p, t''_f)$ received from $\mathcal{F}_{\mathsf{rec}}$. Moreover, every $P_i \in \mathcal{P}'$ outputs $L$ shares of the gate
   outputs $z_i{}^{(k)} = r^{(k)} s^{(k)} + r^{(k)} b_i{}^{(k)} + s^{(k)} a_i{}^{(k)} + c_i{}^{(k)}$, where $r^{(k)}$ and $s^{(k)}$ are received from $\mathcal{F}_{\mathsf{rec}}$.
```

Proof.

Simulator $\mathcal{S}_{\mathsf{mult}}$

```
The simulator interacts with the ideal functionality $\mathcal{F}_{\mathsf{mult}}$ and the adversary $\mathcal{A}$, who expects to interact
(once) with $\mathcal{F}_{\mathsf{rec}}$. The simulator outputs whatever $\mathcal{A}$ outputs.
1: Receive from $\mathcal{F}_{\mathsf{mult}}$ $(\textsc{Ok}, L, \mathcal{P}', t'_a, t'_p, t'_f, (g_r^{(k)}, g_s^{(k)})_{k=1...L}, (a_j^{(k)}, b_j^{(k)}, c_j^{(k)})_{k=1...L, P_j \in (\mathcal{P}_a \cup \mathcal{P}_p)})$.
   // Emulate for $\mathcal{A}$ the interaction with $\mathcal{F}_{\mathsf{rec}}$.
2: Send to $\mathcal{A}$ $(\textsc{Ok}, 2L, \mathcal{P}', t'_a, t'_p, t'_f, (g_r^{(k)})_{k=1...L}, (g_s^{(k)})_{k=1...L})$.
3: Receive from $\mathcal{A}$ a valid configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ with $\mathcal{P}'' \subseteq \mathcal{P}'$. (Set $\mathcal{P}'' = \mathcal{P}'$ and $t''_{\mathsf{x}} = t'_{\mathsf{x}}$ if valid
   values are not received.)
4: Send to $\mathcal{F}_{\mathsf{mult}}$:
   – The configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ received from $\mathcal{A}$,
   – The shares $z_j^{(k)} = g_r^{(k)}(0) \cdot b_j^{(k)} + g_s^{(k)}(0) \cdot a_j^{(k)} + c_j^{(k)}$.
```

We ignore the case when $\mathcal{F}_{\mathsf{mult}}$ executes Complete Break Down. The values sent to $\mathcal{A}$ are exactly the same in the real world and in the simulation (this follows from the way $\mathcal{F}_{\mathsf{mult}}$ computes $g_r^{(k)}$ and $g_s^{(k)}$). Consider now the outputs of parties. The configuration used by $\mathcal{S}_{\mathsf{mult}}$ is trivially the same as the one eventually used by $\mathcal{F}_{\mathsf{rec}}$ in the real world. Hence, both in the real world and in the simulation, all parties output the same configuration $(\mathcal{P}', t'_a, t'_p, t'_f)$, and parties in the same set $\mathcal{P}'$ output shares. To show that the distribution of these shares is the same in both worlds, we show that the sharing polynomials $g_z^{(k)}$ come from the same distributions. The fact that the free coefficients $z^{(k)} = x^{(k)} y^{(k)}$ are the same follows easily from the correctness of circuit randomization. For the other coefficients, observe that the corrupted shares $z_j^{(k)}$ are computed by $\mathcal{S}_{\mathsf{mult}}$ as in the protocol, so in both worlds the polynomials $g_z^{(k)}$ are random among those with degree d, free coefficient $x^{(k)}y^{(k)}$ and going through points $(\alpha_j, z_j^{(k)})$. □

8.6 Realizing $\mathcal{F}_{\mathsf{rec}}$

Finally, we present the protocol ReconPubRobust that realizes $\mathcal{F}_{\mathsf{rec}}$ by invoking the detectable protocol RecPub within the player elimination framework.

Protocol ReconPubRobust $(L, (\mathcal{P}', t'_a, t'_p, t'_f), [s_1]_d, \ldots, [d_L]_d)$

```
Let $\ell = \lceil \frac{L}{t_a+t_f} \rceil$ and $d = t_a + t_p$.
Set $\mathcal{P}'' = \mathcal{P}'$, $t''_a = t'_a$, $t''_p = t'_p$ and $t''_f = t'_f$. For each segment $k = 1 \ldots (t_a + t_f)$ of sharings $[\hat{s}_1]_d, \ldots, [\hat{s}_\ell]_d$ do:
 1: Every party in $\mathcal{P}'$ sets its happy-bit to “happy”.
    // Detectable Computation
 2: The parties RecPub $(d, \ell, (\mathcal{P}'', t''_a, t''_p, t''_f), [\hat{s}_1]_d, \ldots, [\hat{s}_\ell]_d)$.
    // Fault Detection
 3: The parties invoke FaultDetect. If as a result they are all “happy”, the parties in $\mathcal{P}''$ store the reconstructed values and continue to the next segment. Otherwise, they proceed as follows.
    // Fault Localization
 4: Let $P_r$ be the party with the smallest index in $\mathcal{P}''$. Every $P_i \in \mathcal{P}''$ sends to $P_r$ all shares it inputted to RecPub in this segment and the messages $M_i$ it received during RecPub and FaultDetect.
 5: For each sharing inputted to RecPub, $P_r$ uses the received shares to reconstruct the sharing polynomial and the correct shares of all parties.[15] Then, for each $P_i$, $P_r$ reproduces all messages that $P_i$ should have sent, using the reconstructed shares and $M_i$, and, for each $P_j$, compares them with the messages $P_j$ claims to have received from $P_i$ (as specified in $M_j$). $P_r$ broadcasts a tuple $(l, P_i, P_j, x, x')$, such that the $l^{th}$ message $P_j$ claims to have received from $P_i$ was $x'$, while it should have been $x$.[16]
 6: $P_i$ broadcasts whether it agrees with $P_r$ (i.e., whether the $l^{th}$ message it sent to $P_j$ was $x$). $P_j$ broadcasts whether it agrees with $P_r$ (i.e., whether if the $l^{th}$ message it received from $P_i$ was $x'$).
 7: If $P_i$ disagrees, every party sets $E = \{P_i, P_r\}$. If $P_j$ disagrees, every party sets $E = \{P_j, P_r\}$. Otherwise, every party sets $E = \{P_i, P_j\}$.
 8: For every $P_h \in \{P_i, P_j, P_k\}$, the parties in $\mathcal{P}''$ invoke Heartbeat$(P_h)$.
    // Player Elimination
 9: if the output of every invocation of Heartbeat is “alive” then
10:     Every party sets $\mathcal{P}'' = \mathcal{P}'' \setminus E$, $n'' = n'' - 2$ and $t''_a = t''_a - 1$.
11: else
12:     Denote the set of parties, for whom the output of Heartbeat was “crashed” by $E$. Every party sets $\mathcal{P}'' = \mathcal{P}'' \setminus E$, $n'' = n'' - |E|$ and $t''_f = t''_f - |E|$.
13: end if
14: The parties repeat from Step 2.
Every $P_i$ outputs the configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$. Moreover, every $P_i \in \mathcal{P}''$ outputs the $L$ reconstructed values, stored in Step 3.
```

Theorem 4. *Assuming that $3t_a + 2t_p + t_f < n$, the protocol* ReconPubRobust *securely evaluates $\mathcal{F}_{\mathsf{rec}}$ in the presence of a static mixed adversary. The communication complexity of* ReconPubRobust *is $O(Ln\kappa + (t_a + t_f)(n^2\kappa + \mathcal{BA}(1)))$.*

[15] The sharing polynomial can be reconstructed using error correction, for example, the Berlpkamp-Welch algorithm. If $d < n' - 2t'_a - t'_f$, then this reconstruction is correct, which follows by the same argument as in Lemma 7.

[16] If there was no such tuple, then after FaultDetect all parties would have been happy.

Proof. It is enough to show correctness, since $\mathcal{F}_{\mathsf{rec}}$ is deterministic and it reveals all inputs to the adversary (so the simulator is trivial).

Assume that $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid (otherwise the functionality would execute Complete Break Down). Then, $d = t_a + t_f < n' - 2t'_a - t'_f$. The fact that this is preserved throughout the protocol (i.e., correctness of fault detection and localization, and player elimination) follows from the same arguments as those in the proof of Theorems 2 and 1. Hence, the resulting configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ is valid and, by Lemma 8, the sharings are correctly reconstructed. COMPLEXITY: The complexity of detectable computation together with fault detection is $O(\ell n\kappa + n^2\kappa + \mathcal{BA}(1))$, which follows by Lemmas 1 and 8. The complexity of fault localization is the same as above with the addition of $O(\mathcal{BA}(\kappa))$, since the parties send all messages they received and then P_r broadcasts two field elements (the rest of the messages does not affect the asymptotic complexity).

Since the adversary can make the parties repeat a segment at most $t_a + t_f$ times, the overall complexity is $O(Ln\kappa + (t_a + t_f)(n^2\kappa + \mathcal{BA}(1)))$.

□

8.7 Complexity

The communication complexity of evaluation phase (without executing PreparationPhase) is $O((c_i + c_m + c_o)n\kappa + c_i\mathcal{BA}(\kappa) + D(n^3\kappa + n\mathcal{BA}(\kappa)))$ bits, where D is the multiplicative depth of the circuit. For the analysis, consider the individual phases of MPC:

INPUT: This phase communicates $O(c_i(n\kappa + \mathcal{BA}(\kappa)))$ bits, which follows by Lemma 6.

COMPUTATION: Evaluating L multiplication gates (one call to $\mathcal{F}_{\mathsf{mult}}$) communicates $O(Ln\kappa + (t_a + t_f)(n^2\kappa + \mathcal{BA}(1)))$, which follows by Theorem 4. Hence, the cost of all multiplications is $O(c_m n\kappa + D(t_a + t_f)(n^2\kappa + \mathcal{BA}(1)))$, where D is the multiplicative depth of the circuit (note that the multiplication protocol is called at most D times). This amount to $O(c_m n\kappa + D(n^3\kappa + n\mathcal{BA}(1)))$

OUTPUT: This phase communicates $O(c_o n\kappa)$ bits, which follows by Lemma 6.

9 Conclusions

In this paper we present a perfectly-secure protocol which allows to securely evaluate any circuit in the presence of a mixed adversary. Our protocol has linear communication complexity per multiplication gate. Furthermore, our protocol is secure as long as $3t_a + 2t_p + t_f < n$, which is optimal. Previous results for specific types of corruption can be seen special cases of our result: we immediately get a linear protocol for the active setting (with $t_p = t_f = 0$ and $3t_a < n$), a linear protocol for the passive setting (with $t_a = t_f = 0$ and $2t_p < n$) and a linear protocol for a fail-stop adversary (for $t_f < n$). Moreover, our result is much more general and implies protocols with linear complexity for any combination of these settings. For example, we achieve a protocol, where overall 2/3 of the parties are corrupted.

Furthermore, we present a precise, simulation-based proof of security. As a special case, this also yields the first actively-secure protocol with linear complexity and a sound simulation-based proof (note the the proof in [BH08] is property-based).

Universal composition. Katz et el. [KMTZ13] formalize synchronous computation with secure channels and guaranteed termination in the UC framework [Can01] and prove that any protocol that realizes a functionality F in the stand-alone model and has a straightline black-box simulator, immediately yields a protocol (with one additional synchronization round) that UC-realizes F. We note that they consider the active-only setting, so their result cannot be readily applied to our protocol for the mixed setting. However, we believe that with minor extensions, a similar result can be phrased for our formulation of the mixed setting. We remark that we chose to use the stand-alone model, since it naturally models the setting of synchronous computation over secure channels, which results in simpler proofs. The communication in the UC framework, on the other hand, is asynchronous and over insecure channels.

Hence, to prove our protocol secure in the UC framework, we would need techniques similar to those of [KMTZ13], where the authors use the hybrid model with clock and secure channels functionalities.

Future work. An interesting direction for future work is to consider efficient protocols tolerating a mixed adversary in the cryptographic and in the statistically-secure setting (hence, with larger thresholds). Another important problem is to construct a protocol for the mixed setting, that does not have the additive factor $D \cdot p(n)$ in the communication complexity. This can potentially be achieved by combining our techniques with those of [GLS19].

Appendix

A Further Preliminaries

A.1 Hyper-Invertible Matrices

Hyper-invertible matrices were introduced in [BH08], as a key technique to achieve perfectly-secure protocols with linear communication complexity. A matrix is hyper-invertible if its every square sub-matrix is invertible. We refer to [BH08] for a construction.

Definition 3 ([BH08]). *An m-by-n matrix M is hyper-invertible if for any sets of indices $I \subseteq \{1, \ldots, m\}$ and $J \subseteq \{1, \ldots, n\}$ with $|I| = |J| > 0$, the matrix consisting of rows $i \in I$ and columns $j \in J$ of M is invertible.*

Hyper-invertible matrices allow the parties to simultaneously generate a number of consistent sharings of random values, unknown to the adversary. Intuitively, we start with every party providing a (possibly inconsistent) sharing of a random value. Then, we apply to this vector of sharings a hyper-invertible matrix. Given that at most t of the original sharings are possibly known to the adversary, it can be shown that any subset of $n - t$ output sharings is uniformly random and unknown to the adversary. Moreover, given that at most t of the original sharings are possibly inconsistent, it can be shown that verifying the consistency of only t of the output sharings is sufficient to ensure that *all* sharings must be consistent. The above properties are implied by the following lemma.

Lemma 5 ([BH08]). *Let M be a hyper-invertible n-by-n matrix and let $(r_1, \ldots, r_n)^T = M(s_1, \ldots, s_n)^T$. Then for any sets of indices $I, J \subseteq \{1, \ldots, n\}$ with $|I| + |J| = n$, there exists a bijective affine function $f : \mathbb{F}^n \to \mathbb{F}^n$, mapping the values $\{s_i\}_{i \in I}$, $\{r_j\}_{j \in J}$ to $\{s_i\}_{i \notin I}$, $\{r_j\}_{j \notin J}$.*

A.2 Circuit Randomization

In order to evaluate multiplication gates, we use the circuit-randomization technique proposed by Beaver [Bea92]. Namely, for every multiplication gate, the parties pre-compute a triple of sharings $([a]_d, [b]_d, [c]_d)$, where a and b are uniformly random and unknown to the adversary, and $c = ab$. Later, when the parties want to compute the product of two shared values $[x]_d$ and $[y]_d$, they proceed as follows: first, they locally compute $[r]_d = [x]_d - [a]_d$ and $[s]_d = [y]_d - [b]_d$. Then, the randomized sharings $[r]_d$ and $[s]_d$ are publicly reconstructed and the sharing of the product is computed as $[z]_d = rs + r[b]_d + s[a]_d + [c]_d$. This works, because $z = xy = (x - a)(y - b) + (x - a)b + (y - b)a + ab$, and because the values r and s perfectly blind the real inputs x and y.

A similar technique can be used to evaluate an input gate: the parties pre-compute a sharing $[a]_d$ of a random value a. This value is then reconstructed towards the party with the input x who broadcasts the blinded input $r = x - a$, and the parties compute the sharing of x as $[x]_d = r + [a]_d$.

B Secret Reconstruction

This section considers various secret reconstruction protocols that work with sharings of arbitrary degrees. Depending on the degree, the protocols are robust or only detectable.

B.1 Private Reconstruction

The standard protocols RecPriv and RecPrivRobust allow to reconstruct a sharing towards a party P_r. RecPriv detectably reconstructs a sharing $[s]_d$ towards $P_r \in \mathcal{P}'$, given that the sharing degree $d < n' - t'_a$. RecPrivRobust robustly reconstructs a sharing $[s]_d$ towards $P_r \in \mathcal{P}$, assuming that $d < n' - 2t'_a - t'_f$.[17]

Protocol RecPriv$(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$

```
1: Every $P_i \in \mathcal{P}'$ sends its share $s_i$ to $P_r$.
2: If there exists a polynomial $g$ of degree at most $d$ such that all $n'$ received shares lie on $g$, then $P_r$
   outputs $s = g(0)$. Otherwise, $P_r$ becomes unhappy.
```

Protocol RecPrivRobust$(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$

```
1: Every $P_i \in \mathcal{P}'$ sends its share $s_i$ to $P_r$, who stays happy even if there are parties from which it does
   not receive shares.
2: $P_r$ finds a polynomial $g$ of degree at most $d$ such that at least $d + t'_a + 1$ of the received shares lie on
   $g$ (using, for example, the Berlekamp-Welch algorithm), and computes $s = g(0)$.
```

Lemma 6. *Assuming that $d' < n' - t'_a$, $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid and the values inputted by the parties are consistent,* RecPriv *detectably reconstructs the sharing $[s]_d$ towards $P_r \in \mathcal{P}'$.*
The communication complexity of RecPriv *is $O(n'\kappa)$.*

Proof. Assume that no fail-stop corrupted party crashed and that P_r ends up happy. This means that the polynomial g computed by P_r is such that n' values sent in Step 1 lie on g. Since $d < n' - t'_a$, g is fully determined by the $n' - t'_a$ values received by P_r from correct parties. Hence, the output of P_r is correct.

If all parties behave according to the protocol, correctness of RecPriv is trivial. Privacy is trivially preserved as well. The complexity follows by inspection of the protocol. ☐

Lemma 7. *Assuming that $d < n' - 2t'_a - t'_f$, , $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid and the values inputted by the parties are consistent,* RecPrivRobust *robustly reconstructs the sharing $[s]_d$ towards $P_r \in \mathcal{P}$.*
The communication complexity of RecPrivRobust *is $O(n'\kappa)$.*

Proof. Assume that $d < n' - 2t'_a - t'_f$. Since at least $d + t'_a + 1$ of the values received by P_r lie on g and at most t'_a of these values can come from actively corrupted parties, g is determined by the at least $d + 1$ shares sent by correct parties. Furthermore, P_r always receives at least $n' - t'_a - t'_f$ values from correct parties. Those values lie on g. Since, by the assumption, $d + t'_a + 1 \leq n' - t'_a - t'_f$, P_r always receives at least $d + t'_a + 1$ values lying on g.

The complexity follows by inspection of the protocol. ☐

B.2 Public Reconstruction

The goal of the public reconstruction protocol RecPub is to reconstruct a number of sharings towards all parties in $\mathcal{P}'$. RecPub is detectable and assumes that the degree of the sharing polynomial is $d < n' - t'_a$.

RecPub is based on the idea of [DN07] to reconstruct a bucket of sharings, using error correcting codes. Intuitively, $n' - t'_a$ sharings are expanded to n' sharings, using a fixed linear error correcting code. Each of the n' resulting sharings is opened towards a different party in $\mathcal{P}'$. Then, the parties exchange the reconstructed values, and each of them checks whether the received values form a codeword.
RecPub reconstructs any number ℓ of sharings by invoking the above procedure a number of times (in parallel).

[17] Since RecPrivRobust will be used to evaluate output gates and all parties in $\mathcal{P}$ can receive outputs, we need to be able to reconstruct a value towards any party in $\mathcal{P}$.

Protocol $\mathsf{RecPub}(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f), [s_1]_d, \dots, [s_\ell]_d)$

The sharings are processed in buckets of size $n' - t'_a$ (with the last bucket possibly smaller). For each bucket $[\hat{s}_1]_d, \dots, [\hat{s}_l]_d$ of size $l \leq n' - t'_a$ do the following:

```
1: The parties compute locally $([u_1]_d, \dots, [u_{n'}]_d)^T = V([\hat{s}_1]_d, \dots, [\hat{s}_l]_d)^T$, where $V$ is the Vandermonde matrix of size $n' \times l$ defined by some fixed vector $\beta$ with unique values.
2: For $i \in \{1, \dots, n'\}$, the parties invoke RecPriv to reconstruct $[u_i]_d$ towards $P_i$.
3: Every $P_i \in \mathcal{P}'$ sends $u_i$ to every $P_j \in \mathcal{P}'$.
4: Every $P_j \in \mathcal{P}'$ checks whether there exists a polynomial $g$ of degree at most $l-1$ such that all points $(\beta_1, u_1), \dots, (\beta_{n'}, u_{n'})$ lie on $g$. If this is the case, $P_i$ takes as the reconstructed values $\hat{s}_1, \dots, \hat{s}_l$ the $l$ coefficients of $g$. Otherwise, $P_j$ gets unhappy.
```

Every happy $P_i \in \mathcal{P}'$ outputs the tuple $s_1, \dots, s_\ell$, which is the concatenation of the values $\hat{s}_1, \dots, \hat{s}_l$ from all buckets.

The communication cost of RecPub can be seen to be $O(\ell n' \kappa + n'^2 \kappa)$. Hence, for large enough ℓ, the amortized complexity of reconstructing one sharing is $O(n' \kappa)$.

Lemma 8. *Assuming that $d < n' - t'_a$, $(\mathcal{P}', t'_a, t'_p, t'_f)$ is valid and the values inputted by the parties are consistent,* RecPub *detectably reconstructs ℓ sharings $[s_1]_d, \dots, [s_\ell]_d$ towards all parties in $\mathcal{P}'$. The communication complexity of* RecPub *is $O(\ell n' \kappa + n'^2 \kappa)$.*

Proof. DETECTABILITY: We prove that after RecPub, every correct party $P_j \in \mathcal{P}$ either gets the correct output or is unhappy.

Consider one bucket $[\hat{s}_1]_d, \dots, [\hat{s}_l]_d$ of sharings, where $l \leq n' - t'_a$ and $d < n' - t'_a$. By Lemma 6, in Step 2 every correct P_i reconstructs the correct value u_i (or gets unhappy). Hence, in Step 3 every correct P_j receives at least $n' - t'_a$ correct values u_i from the correct parties (or gets unhappy). The polynomial g of degree at most $l - 1$ computed by P_j in Step 4 is fully determined by these $n' - t'_a \geq l$ correct values u_i. Hence, P_j either reconstructs the correct values $\hat{s}_1, \dots, \hat{s}_l$, or gets unhappy.

COMPLETENESS AND PRIVACY: If all parties behave according to the protocol, correctness of RecPub is trivial. Privacy is not an issue in RecPub.

COMPLEXITY: In order to compute one bucket, the parties communicate $O(n'^2 \kappa)$ bits, which follows by inspection of the protocol. Hence, the communication complexity of all $\lceil \ell / (n' - t'_a) \rceil$ buckets is $O(\frac{\ell}{n' - t'_a} n'^2 \kappa + n'^2 \kappa)$. Moreover, since $3t'_a < n'$, we have $n' - t'_a \geq \frac{2}{3} n'$ and the overall communication complexity of RecPub is $O(\ell n' \kappa + n'^2 \kappa)$.

$\square$

References

[AL17] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly secure multiparty computation. *Journal of Cryptology*, 30(1):58–151, January 2017.

[APC+08] B. V. Ashwinkumar, Arpita Patra, Ashish Choudhary, Kannan Srinathan, and C. Pandu Rangan. On tradeoff between network connectivity, phase complexity and communication complexity of reliable communication tolerating mixed adversary. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *27th ACM PODC*, pages 115–124. ACM, August 2008.

[Bea90] Donald Beaver. Multiparty protocols tolerating half faulty processors. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 560–572. Springer, Heidelberg, August 1990.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BFH+08] Zuzana Beerliová-Trubíniová, Matthias Fitzi, Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. MPC vs. SFE: Perfect security in a unified corruption model. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 231–250. Springer, Heidelberg, March 2008.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 663–680. Springer, Heidelberg, August 2012.

[BGP92] Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. In *Computer science*, pages 313–321. Springer, 1992.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Heidelberg, March 2008.

[BHKL18] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 695–712. ACM Press, October 2018.

[BJMS18] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. Cryptology ePrint Archive, Report 2018/580, 2018. `https://eprint.iacr.org/2018/580`.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

[CDN12] Ronald Cramer, Ivan Damgard, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing—An Information Theoretic Approach.* 2012. Available at `https://www.moodle.ch/lms/pluginfile.php/6253/mod_book/chapter/25/mpc-book.pdf`.

[Cha90] David Chaum. The spymasters double-agent problem: Multiparty computations secure unconditionally from minorities and cryptographically from majorities. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 591–602. Springer, Heidelberg, August 1990.

[CL14] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 466–485. Springer, Heidelberg, December 2014.

[CPA+08] Ashish Choudhary, Arpita Patra, B. V. Ashwinkumar, K. Srinathan, and C. Pandu Rangan. Perfectly reliable and secure communication tolerating static and mobile mixed adversary. In Reihaneh Safavi-Naini, editor, *ICITS 08*, volume 5155 of *LNCS*, pages 137–155. Springer, Heidelberg, August 2008.

[DDWY93] Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. Perfectly secure message transmission. *Journal of the ACM*, 40(1):17–47, January 1993.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, August 2006.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg, May / June 2010.

[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*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[DNPR16] Ivan Damgård, Jesper Buus Nielsen, Antigoni Polychroniadou, and Michael Raskin. On the communication required for unconditionally secure multiplication. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 459–488. Springer, Heidelberg, August 2016.

[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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[FHM98] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multi-party computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 121–136. Springer, Heidelberg, August 1998.

[GIP$^+$14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 85–114. Springer, Heidelberg, August 2019.

[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 ACM STOC*, pages 218–229. ACM Press, May 1987.

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

[GP92] Juan A. Garay and Kenneth J. Perry. A continuum of failure models for distributed computing. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms: 6th International Workshop, WDAG '92 Haifa, Israel*, pages 153–165, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[GP09] Hossein Ghodosi and Josef Pieprzyk. Multi-party computation with omnipresent adversary. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 180–195. Springer, Heidelberg, March 2009.

[HLMR11] Martin Hirt, Christoph Lucas, Ueli Maurer, and Dominik Raub. Graceful degradation in multi-party computation (extended abstract). In Serge Fehr, editor, *ICITS 11*, volume 6673 of *LNCS*, pages 163–180. Springer, Heidelberg, May 2011.

[HMP00] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 143–161. Springer, Heidelberg, December 2000.

[HMZ08] Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. MPC vs. SFE: Unconditional and computational security. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 1–18. Springer, Heidelberg, December 2008.

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 463–482. Springer, Heidelberg, August 2006.

[IKLP06] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On combining privacy with guaranteed output delivery in secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 483–500. Springer, Heidelberg, August 2006.

[Kat07] Jonathan Katz. On achieving the "best of both worlds" in secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 11–20. ACM Press, June 2007.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.