Efficient MPC with a Mixed Adversary

Martin Hirt
ETH Zurich, Switzerland
hirt@inf.ethz.ch

Marta Mularczyk
ETH Zurich, Switzerland
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.

2012 ACM Subject Classification Security and privacy → Network security

Keywords and phrases Multi-party Computation, Communication Cost

Digital Object Identifier 10.4230/LIPIcs.ITC.2020.3

Related Version A full version of the paper is available at https://eprint.iacr.org/2020/356.

Funding Marta Mularczyk: Research supported by the Zurich Information Security and Privacy Center (ZISC).

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. Since the seminal works [9, 15], it is known that a perfectly-secure protocol can only tolerate \( t < n/3 \) active corruptions, or \( t < n/2 \) passive corruptions. 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. [26] 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 \( n/3 \). [26] 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.

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 [9, 15] and in the mixed setting [26] 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 [19, 38, 22, 8, 10, 29, 33], 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. 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.
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.
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 [22] 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 $k = n/3$ 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 [22], 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 [27] 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.\footnote{Note that the fail-corrupted parties do not reveal their inputs, unless they are also passively or actively corrupted.}

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 [8] 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 [8] 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 [35], 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 [35]. 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.

1.2 Related Work

Communication-Efficient Active MPC

Since the seminal feasibility results [45, 32, 9, 15, 43, 5], 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 [19, 38], in the unconditional setting (that is, allowing negligible error probability) [22, 20, 10, 29, 28], and in the setting with perfect security [8, 33].

There also exist efficient protocols that can tolerate a dishonest majority [24, 21, 42, 4] (where the online phase has linear complexity). We stress that such protocols can only offer degraded security [17], in particular, no fairness or guaranteed output delivery. In this work, we focus on full security.

On the negative side, it has been proved [23] 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 and $p$ is a small polynomial. This caveat was recently removed by Goyal et al. [33], 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 [33], 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 [14, 39, 40]. 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. [3] 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 [25, 16, 2], and in the context of degraded security [34].

Ghodosi and Pieprzyk [30] 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 [7, 36] (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 [6], 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 [8], 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 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 [22]). Evaluating an output gate corresponds to secret reconstruction.

Both phases employ the player-elimination framework [35], 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

---

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.
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 [8], 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
\(O(|C| n \kappa + n^3 \kappa + D n^3 \kappa + c_i n^2 \kappa)\), 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. 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 [12] with static adversary and synchronous
communication over perfectly-secure channels (for details of this model, see [31] or [12]). 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

\(^6\) One can easily get linear dependency on the number of input gates, using the techniques of [8]. 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.
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 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. 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 $F$. 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 $F$, 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 $F$ with domain $(X \cup \{\perp\})^n$, where $\perp \not\in X$. 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 $\perp$. $F$ 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 $A$, there exists an ideal-world adversary $S$, such that the joint distribution of the view of $A$ and the outputs of the correct parties executing the protocol is equal to the joint distribution of the output of $S$ and the outputs of the correct parties computed by the ideal functionality $F$.

**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.

**Corruption-aware functionalities**

The model of [12] was extended by Asharov and Lindell [1] 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 $P_a, P_p, P_f$ of corrupted parties. Then, each functionality receives, upon initialization, the same set of parties actually controlled by the

---

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.
adversary. Note that this is well defined in the static setting. We refer to [12] for a formal description. Finally, we note that the corruption-aware functionalities were used in [1] 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 [1] is not corruption aware.

3 Preliminaries

We assume familiarity with hyper-invertible matrices and circuit randomization (we explain these concepts in the full version [37]).

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 \( P \) is reduced to a smaller set, which we denote by \( P' \). Accordingly, we use \( t'_a, t'_p, t'_f \) and \( n' \) to denote the corruption thresholds in \( P' \) and \(|P'|\), respectively. A tuple \((P', t'_a, t'_p, t'_f)\) 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 \( P' \). However, since we use Shamir sharings with degree \( d = t_a + t_p \) (independent of \( P' \)), we also require that such sharings are possible to reconstruct by the parties in \( P' \). 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 \((P', t'_a, t'_p, t'_f)\) is valid if (1) \( P' \subseteq P \), (2) \( t_a + t_p < n' - 2t'_a - t'_f \), and (3) \(|P' \cap P_x| \leq t'_x\) for all \( x \in \{a, p, f\} \), where \( P_x \) 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 \( P' \). 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 [18]). 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 [27]. They include...
consistency – the outputs of all parties that are not actively corrupted are equal, and persistence – if inputs of all correct parties are equal to \( x \), then the outputs are equal to \( x \).

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 \[27\]. 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 \[27\] is \( O(n^4) \). However, by applying the king-simulation technique of \[11\], 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 \( P' \), but all parties in \( P \) should learn the output.\(^{11}\) Hence, we employ the following consensus protocol, where every party \( P'_i \in P' \) has input \( x_i \), and every party in \( P \) receives output. The protocol internally invokes a standard consensus protocol, and it is secure assuming that \( 3t'_a + 2t'_p + t'_f < n' \).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Consensus(({x_i}_{P'_i \in P'}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>The parties in ( P' ) invoke a standard consensus protocol (for example, [27, 11]), where the input of ( P'_i ) is ( x_i ).</td>
</tr>
<tr>
<td>2.</td>
<td>Every party in ( P' ) sends the output to every party in ( P \setminus P' ).</td>
</tr>
<tr>
<td>3.</td>
<td>Every party in ( P' ) outputs the result of the consensus, and every party in ( P \setminus P' ) determines its output using the majority rule.</td>
</tr>
</tbody>
</table>

Moreover, all parties in \( P \) should be able to act as a sender in broadcast.\(^{11}\) Such broadcast can easily be constructed by having a sender \( P_\ell \in P' \) send his value to all parties in \( P' \), who then invoke Consensus on the received values.

The cost of Consensus instantiated with the protocol of \[27, 11\] is \( O(n^2) \) (note that \( n \) is the size of the initial set \( P \)). We denote by \( B_A(\kappa) \) the complexity of broadcasting or reaching consensus on a \( \kappa \)-bit message in the mixed setting. With the above protocol, \( B_A(\kappa) = O(n^2\kappa) \).

\( \triangleright \) 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 \( \bot \).

\(^9\) In terms of \[27\], 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.

\(^{11}\) The reason is that eliminated parties can still provide inputs to the MPC protocol.
The player-elimination framework was introduced in [35] 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.

Unfortunately, the player-elimination framework for the active setting [35] 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 [35] 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 (1) completeness – a protocol should give correct outputs if there are no faults during the execution and (2) privacy, which must be preserved always, no matter if there were crashes or malicious behavior.

Definition 2. A passively secure protocol $\Pi$ for a set of parties $P'$ is detectable if after the execution of $\Pi$ 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 $P'$ is unhappy, or (3) at least one fail-stop corrupted party in $P'$ crashed. Moreover, if all parties are honest, then no (correct) party becomes unhappy. Furthermore, $\Pi$ guarantees privacy in all cases.

Fault Detection

The protocol FaultDetect is executed by the parties in $P'$ 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.

1: Every $P_i \in P'$ 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 $P'$ run consensus on their happy-bits.
3: Every $P_i \in P'$ sets its happy-bit to the result of the consensus.

Lemma 3. 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 $O(n^2 + BA(1))$.

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”.

<table>
<thead>
<tr>
<th>Protocol Heartbeat($P_h, (\mathcal{P}', t'_a, t'_p, t'_f)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $P_h$ sends the value 1 to every party $P_j \in \mathcal{P}'$.</td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

Lemma 4. 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 $O(n + BA(1))$.

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 [44], 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 $(\alpha_i, s_i)$ 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 the full version [37]). 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 $\text{RecPriv}(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$ 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 $\text{RecPrivRobust}(P_r, d, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d)$ 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 $\text{RecPub}(d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f), [s]_d, \ldots, [s]_{\ell d})$ detectably reconstructs $\ell$ sharings towards all parties in $\mathcal{P}'$, assuming $d' < n - t'_a$ 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 [22].
7 Preparation Phase

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

7.1 The Functionality $F_{\text{prepPhase}}$

The parties input to the functionality the desired number $L$ of triples and the current configuration $(P', t'_a, t'_p, t'_f)$. Since the protocol employs player elimination, the resulting triples are shared among the parties in a smaller set $P''$. The resulting configuration $(P'', t''_a, t''_p, t''_f)$ is outputted to all parties. Moreover, all parties in $P''$ 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 $(P'', t''_a, t''_p, t''_f)$, as long as it is valid and decides on the shares of passively and actively corrupted parties.

The functionality receives sets of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$.

1. Receive from each party a number $L$ and a valid configuration $(P', t'_a, t'_p, t'_f)$.
2. If these values (ignoring $\bot$) are not consistent among parties, or if $(P', t'_a, t'_p, t'_f)$ is not valid, execute Complete Break Down. Else, send $(\text{Ok}, L, P', t'_a, t'_p, t'_f)$ to the adversary and proceed as follows.
3. The adversary sends:
   - A valid set $P'' \subseteq P'$ with thresholds $t''_a, t''_p, t''_f$.
   - For each $P_j \in (P_a \cup P_p)$, shares $(a_j^{(k)}, b_j^{(k)}, c_j^{(k)})_{k=1,\ldots,L}$.

(For $P'' = P'$, $t''_a = t'_a$ and $(a_j^{(k)}, b_j^{(k)}, c_j^{(k)}) = (0, 0, 0)$ if valid values are not received.)
4. Send $(P'', t''_a, t''_p, t''_f)$ to all parties.
5. Generate $L$ triples shared among the parties in $P''$ as follows. For each triple $k$, choose random $d^{(k)}$ and $b^{(k)}$, and let $c^{(k)} = d^{(k)} b^{(k)}$. Then, for each $k = 1, \ldots, L$ and $x \in \{a, b, c\}$, choose the polynomial $g_x^{(k)}$ at random from the set of all polynomials of degree at most $d$, going through the point $(0, x^{(k)})$ and all points in $\{(a_j, x_j^{(k)}) | P_j \in (P_a \cup P_p)\}$. Send to each $P_i \in P''$ its shares $(g_a^{(k)}(a_i), g_b^{(k)}(a_i), g_c^{(k)}(a_i))$.

7.2 The Functionality $F_{\text{triples}}$

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 $F_{\text{triples}}$ the desired number of triples $\ell$ and the configuration $(P', t'_a, t'_p, t'_f)$. Then, the adversary decides on one of three outcomes: (1) The detectable computation succeeds, and $\ell$ triples shared with degree $d = t_a + t_p$ among the parties in $P'$ 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).
3:14 Efficient MPC with a Mixed Adversary

The functionality receives sets of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$.

1. Receive from each party a number $\ell$ and a valid configuration $(P', t_a', t_p', t_f')$.
2. If these values (ignoring $\perp$) are not consistent among parties, or if $(P', t_a', t_p', t_f')$ is not valid, execute Complete Break Down. Else, send $(Ok, \ell, P', t_a', t_p', t_f')$ to the adversary and proceed as follows.

3. Receive from the adversary a message $M$, which is processed as follows:
   - If $M = (\text{Triples}, ((a_j^{(k)}, g_j^{(k)}, c_j^{(k)}))_{k=1...\ell, P_i \in (P_a \cup P_p)})$, then generate $\ell$ triples shared among the parties in $P'$ as follows. For each triple $k$, choose random $a^{(k)}$ and $b^{(k)}$, and let $c^{(k)} = a^{(k)}b^{(k)}$. Then, for each $k = 1, \ldots, \ell$ and $x \in \{a, b, c\}$, choose the polynomial $g^{(k)}_x$ at random from the set of all polynomials of degree at most $d$, going through the point $(0, x^{(k)})$ and all points in $\{(\alpha_j, x^{(k)}_j) \mid P_j \in (P_a \cup P_p)\}$. Send to each $P_i \in P'$ its shares $(g^{(k)}_x(\alpha_i), g^{(k)}_b(\alpha_i), g^{(k)}_c(\alpha_i))$.
   - If $M = (\text{ActiveSet}, E)$, where $E \subseteq P'$ is a set such that $|E \cap P_a| \geq |E|/2$, then send $(\text{ActiveSet}, E)$ to the parties.
   - If $M = (\text{CrashSet}, E)$, where $E \subseteq P', E \subseteq P_f \cup P_a$ and $E \neq \emptyset$, then send $(\text{CrashSet}, E)$ to the parties.
   - Any other message is treated as $(\text{Triples}, ((0, 0, 0))_{k=1...\ell, P_i \in (P_a \cup P_p)})$.

7.3 Realizing $F_{\text{prepPhase}}$ in the $F_{\text{triples}}$-hybrid Model

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

Let $\ell = \lceil \frac{L}{t_a + t_f} \rceil$ and $d = t_a + t_p$.

Set $P'' = P'$, $t_{a''} = t_{a'}$, $t_{p''} = t_{p'}$ and $t_{f''} = t_{f'}$. For each segment $k = 1 \ldots (t_a + t_f)$ do:

1. Send $\ell$ and $(P'', t_{a''}, t_{p''}, t_{f''})$ to $F_{\text{triples}}$.
2. If the output is $(\text{ActiveSet}, E)$, set $P'' = P'' \setminus E$ and $t_{a''} = t_{a''} - |E|/2$, and repeat Step 1.
3. Else if the output is $(\text{CrashSet}, E)$, then set $P'' = P'' \setminus E$ and $t_{f''} = t_{f''} - |E|$, and repeat Step 1.
4. Else, store the sharings received from $F_{\text{triples}}$ and continue to the next segment.

Every $P_i$ outputs the configuration $(P'', t_{a''}, t_{p''}, t_{f''})$. Moreover, every $P_i \in P''$ outputs the first $L$ triples of shares received from $F_{\text{triples}}$.

**Theorem 5.** Assuming that $3t_a + 2t_p + t_f < n$, the protocol $\text{PreparationPhase}$ securely realizes $F_{\text{prepPhase}}$ in the $F_{\text{triples}}$-hybrid model, in the presence of a static mixed adversary.

**Proof.** The simulator $S_{\text{prepPhase}}$ only needs to simulate the interaction with the ideal functionality $F_{\text{triples}}$. This is done by simply executing the code of $F_{\text{triples}}$.

Throughout the execution in the real world, we have the following invariant: $(P'', t_{a''}, t_{p''}, t_{f''})$ is valid and the shares stored by parties in $P''$ form correct $d$-sharing 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 $F_{\text{triples}}$). This also implies that $3t_a + 2t_p + t_f < n''$. For the latter, notice that a $d$-sharing in $P'$ is also a correct $d$-sharing in $P''$, since $d$ is constant. The above invariant shows that the outputs are the same in the real and in the ideal world.  

12 Observe that we do not need that in case $F_{\text{triples}}$ sends $(\text{CrashSet}, E)$, the parties in $E$ are actually not correct. It is enough that they are in $P_f$.  

7.4 Realizing \( F_{\text{triples}} \)

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 \( F_{\text{triples}} \).

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 \(P_l\), generates a (possibly inconsistent) \(d\)-sharing of \(s\).

\[ \textbf{Protocol Share}(P_l, d, s, (P', t'_a, t'_p, t'_f)) \]

\(P_l\) chooses a random polynomial \(g\) of degree \(d\) and sends to every party \(P_j \in P'\) 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 \(t'_a\) 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 \(t_a + t_p\) 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 \(\ell\) of double sharings, the procedure sketched above is invoked a number of times (in parallel).

\[ \textbf{Protocol DoubleShareRandom}(d, d', \ell, (P', t'_a, t'_p, t'_f)) \]

The \(\ell\) 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 \(P_i \in P'\) chooses \(s_i\) at random and double-shares it among \(P'\) by invoking Share\((P_i, d, s_i, (P', t'_a, t'_p, t'_f))\) and Share\((P_i, d' + s_i, (P', t'_a, t'_p, t'_f))\).
2. The parties compute locally \(\left([r_1]_{d,d'}, \ldots, [r_{n' + 1}]_{d,d'}\right)^T = M([s_1]_{d,d'}, \ldots, [s_{n'}]_{d,d'})^T\), 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 \(P_i \in P'\) and \(j \in \{1, \ldots, 2t'_a\}\), \(P_i\) sends its shares of \([r_j]_{d,d'}\) to \(P_j\).
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 \(\ell\) sharings generated in this bucket are \([r_{n' + 1}]_{d,d'}, \ldots, [r_{n'}]_{d,d'}\).

Output all sharings generated in all buckets (that is, \(\ell\) sharings in total).
The communication cost of DoubleShareRandom can be seen to be $O(ℓn′κ + n^2κ)$. Hence, for large enough $ℓ$, the amortized complexity of generating one double-sharing is $O(n′κ)$.

Lemma 6. Assuming that $(P^i, t_i′, t′_p)$ 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 $O(ℓn′κ + n^2κ)$.

Proof. Consider one bucket, in which $l$ $(d, d′)$-sharings are generated, where $l ≤ n′ − 2t_i′ − t′_p − \min(t_i′, t′_p)$.

Correctness: Assume that no crash occurred and that all the protocol sharings are correct. All values $s_i$ generated by the correct parties in Step 1 are double-shared correctly. Thus, at least $n′ − 2t_i′$ sharings $[s_i]_{d,d′}$ are correct. Moreover, at least $t_i′$ out of $2t_i′$ 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: $t_i′ + t′_p$ values $s_i$ (those chosen by passively or actively corrupted parties) and $\min(2t_i′, t_i′ + t′_p) = t_i′ + \min(t_i′, t′_p)$ values $r_i$ (those possibly revealed to such parties). With these $2t_i′ + t_i′ + \min(t_i′, t′_p)$ values fixed, there is a bijective mapping between the actual $l ≤ n′ − 2t_i′ − t′_p − \min(t_i′, t′_p)$ 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 $O(n^2κ)$ bits, which follows by inspection of the protocol. Moreover, each bucket (but the last one) contains $n′ − 2t_i′ − t′_p − \min(t_i′, t′_p)$ double-sharings, which can be seen to be at least $\frac{1}{2}n′$ \footnote{If we set $m = 3t_i′ + 2t′_p$, then $t_i′ ≥ m/5$ or $t′_p ≥ m/5$. It follows that $n′ − 2t_i′ + t′_p − \min(t_i′, t′_p) = n′ − m + t_i′ + t′_p − \min(t_i′, t′_p) = n′ − m + \max(t_i′, t′_p) + \min(t_i′, t′_p) − \min(t_i′, t′_p) ≥ n′/5.$}, hence, there are at most $5ℓ/n′ + 1$ buckets. Therefore, the overall communication complexity of DoubleShareRandom is $O((5ℓ/n′ + 1)n^2κ) = O(ℓn′κ + n^2κ)$.

Detectably 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 $3t_i′ + 2t′_p + t′_f < n′$. 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 $d′ = t_i′ + t′_p$. 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 $r_i$. 

The following protocol Realizing \( \mathcal{F}_{\text{triples}} \) realizes \( \mathcal{F}_{\text{triples}} \).

### Protocol Realizing \( \mathcal{F}_{\text{triples}} \)

1. Every party in \( \mathcal{P}' \) sets its happy-bit to “happy”.
2. The parties invoke \textbf{GenerateTriples}(\( d, \ell, (\mathcal{P}', t'_a, t'_p, t'_f) \)) for \( d = t_a + t_p \).
3. The parties invoke \textbf{FaultDetect}. If, as a result, they are all happy, they output the generated triples. Otherwise, they do the following.
4. Let \( P_i \) be the party with the smallest index in \( \mathcal{P}' \). Every \( P_i \in \mathcal{P}' \) sends to \( P_i \) the randomness \( R_i \) it used in the two previous steps and the messages \( M_i \) it received in those steps. If \( P_i \) does not receive values from some parties, it uses instead the default values and proceeds.\(^{14}\)
5. For each \( P_i, P_j \) 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, \( P_i \) broadcasts a tuple \((l, P_j, P_j, x, x')\), 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 \( P_j \) (i.e., whether the \( l^{th} \) message it sent to \( P_j \) was \( x \)). \( P_j \) broadcasts whether it agrees with \( P_i \) (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 \( E = \{ P_i, P_r \} \).
8. For every \( P_h \in \{ P_i, P_j, P_r \} \), the parties in \( \mathcal{P}' \) invoke \textbf{Heartbeat}(\( P_h \)).
9. If the output of every invocation of \textbf{Heartbeat} is “alive”, output \( E \). Otherwise, output the set of parties, for whom the output of \textbf{Heartbeat} was “crashed”.

\(^{14}\)If \( P_i \) does not get the values from some party \( P_j \), it could, instead of ignoring it, broadcast the index of such party. However, such solution would make the description of the protocol more involved.
Theorem 8. Assuming that $3t_a + 2t_p + t_f < n$, the protocol Triples securely evaluates $\mathcal{F}_{\text{Triples}}$ in the presence of a static mixed adversary.

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

Assume that the honest parties input consistent values $\ell$, $\mathcal{P}'$, $t'_a$, $t'_p$ and $t'_f$, and that the set $\mathcal{P}'$ is valid (otherwise, the simulation is trivial). This means that the preconditions for Lemmas 3, 4 and 7 are met. By consistency of FaultDetect (Lemma 3), $S_{\text{Triples}}$ does not abort in Step 3. Then, $S_{\text{Triples}}$ sends the command Triples if and only if the result of FaultDetect is “happy”, which is the same as in the real world. Hence, we can consider two cases: (1) the simulator sends Triples, and (2) he sends ActiveSet or CrashSet.

Case Triples. Consider any triple and the corresponding sharing polynomials $g_a$, $g_b$ and $g_c$. In the real world, by correctness of FaultDetect, no party is crashed and no correct party is unhappy at the end of GenerateTriples. Therefore, the secrecy stated in Lemma 7 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 7 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 7), 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 ActiveSet or 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 $S_{\text{Triples}}$ is the same as in the protocol.

The simulator sends CrashSet if and only if the output of Heartbeat for some parties in $\mathcal{P}'$ was not “alive”. In this case, $E$ contains no correct party, by completeness of Heartbeat (Lemma 4).

Now consider the other case, where $S_{\text{Triples}}$ sends ActiveSet. By correctness of 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 Triples. By Lemmas 7 and 3, executing \texttt{GenerateTriples} and \texttt{FaultDetect} requires communicating $O(\ell n \kappa + n^2 \kappa + BA(1))$ 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 \texttt{GenerateTriples} and \texttt{FaultDetect}, and that, asymptotically, sending the values $R_i$ does not add to this complexity. Together with broadcasts and \texttt{Heartbeat}, this results in $O(\ell n \kappa + n^2 \kappa + BA(\kappa))$ bits for the whole protocol Triples.

For the final protocol \texttt{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 protocol \texttt{PreparationPhase} communicates \texttt{PreparationPhase} is $O(L n \kappa + (t_a + t_f)(n^2 \kappa + BA(\kappa)))$, which amounts to $O(L n \kappa + n^3 \kappa)$.

8 Evaluating Any Circuit

In the full version of this paper [37] we show how to use the preprocessing functionality constructed in the previous section to evaluate any circuit. In order to do so, we first define a multiplication functionality $F_{\text{mult}}$, which can be used to evaluate a number of multiplication gates on the same depth. $F_{\text{mult}}$ expects as input a number of shared multiplication triples. We then construct a protocol that realizes the circuit-evaluation functionality $F_{\text{mpc}}$ in the ($F_{\text{prepPhase}}, F_{\text{mult}}$)-hybrid model. Finally, we show how to realize $F_{\text{mult}}$, using circuit randomization, the detectable public reconstruction protocol RecPub and player elimination.

The final result is the following functionality $F_{\text{mpc}}$. We note that since the adversary can always prevent fail-corrupted parties from giving input, $F_{\text{mpc}}$ is defined on an extended input domain, where a party can input $\bot$, meaning that it gives no input. $F_{\text{mpc}}$ replaces the inputs $\bot$ by default inputs 0 and sends to all parties the set of all parties who inputted $\bot$ (this way, parties know what circuit was actually evaluated).

<table>
<thead>
<tr>
<th>Functionality $F_{\text{mpc}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.)</td>
</tr>
<tr>
<td>2: For each party whose input was $\bot$, set all inputs to $C$ to 0.</td>
</tr>
<tr>
<td>3: Evaluate $C$ and send outputs to the corresponding parties. Send to all parties the set of parties who inputted $\bot$.</td>
</tr>
</tbody>
</table>

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 [8] is property-based).

**Universal Composition**

Katz et al. [41] formalize synchronous computation with secure channels and guaranteed termination in the UC framework [13] 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 [41], 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 [33].

**References**


Efficient MPC with a Mixed Adversary


