Diss. ETH No. 17993

Efficient Multi-Party Computation with Information-Theoretic Security

A dissertation submitted to

ETH ZURICH

for the degree of
Doctor of Sciences

presented by

Zuzana Beerliová-Trubíniová
Dipl. Math ETH

born June 26, 1977, in Bratislava
citizen of Slovakia

accepted on the recommendation of

Prof. Dr. Ueli Maurer, examiner
Prof. Dr. Ran Canetti, co-examiner

2008

Acknowledgments

I would like to thank my supervisor Ueli Maurer for giving me the opportunity to work in this fascinating field of cryptography, for his support and for creating a motivating and enjoyable working environment. I also thank Ran Canetti for accepting to be my co-referee and for his helpful comments.

All results presented in this theses originated in the countless passionate discussions with my co-supervisor Martin Hirt. I thank him for the competent guidance and encouragement and for becoming much much more than my co-supervisor.

Many thanks go to Vassilis Zikas for being a nice roommate, for many fruitful discussions, and for proofreading the theses.

I enjoyed my time in our crypto-group very much and would like to thank all former and present members (not only for being good table soccer partners): Divesh Aggarwal, Georges Baatz, Marc Fischlin, Ghislain Fourny, Peter Gaži, Thomas Holenstein, Robert König, Christoph Lucas, Jun Muramatsu, Jesper Nielsen, Krzysztof Pietrzak, Bartosz Przydatek, Dominik Raub, Renato Renner, Junji Shikata, Johan Sjödin, Björn Tackmann, Stefano Tessaro, Douglas Wikström and our secretary Beate Bernhard.

Last but not least I thank my family and my friends for their support and encouragement, especially my father for awakening my love for mathematics and Stefan for bringing me back to it.

Abstract

Multi-party computation (MPC) enables a set of n mutually distrusting players to perform some computation on their private inputs, such that the correctness of the output as well as the privacy of the honest players' inputs is guaranteed even in the presence of an adversary corrupting up to t of the players and making them misbehave arbitrarily.

In this thesis, we focus on the *efficiency* of multi-party computation protocols, and present the following contributions:

- The main efficiency bottleneck in MPC is the computation of the multiplication gates. However, since recently, for $t < n/3$ multiplication can be reduced to non-robust generation of correct sharings of secret random values. We present a novel technique which allows to perfectly and very efficiently generate such random sharings. Based on this technique, we construct a perfectly secure MPC protocol for $t < n/3$, communicating $\mathcal{O}(n\kappa)$ bits per multiplication (where κ denotes the bit-length of a value).

- An efficient way of limiting the number of times when the adversary can disturb (and slow down) the computation is player elimination – every time an inconsistency is detected, a pair of players, at least one of them corrupted, is localized and eliminated. However, honest players can get eliminated as well, which causes several limitations of this technique. We generalize and extend the player elimination-technique, by finding other means (than elimination) of preventing localized players from disturbing the computation ever again. Our new technique, called *dispute control*, allows to construct efficient MPC protocols in settings where player elimination is not applicable: We present an actively secure MPC protocol that provides optimal security (unconditional security against a faulty minority) and communicates only $\mathcal{O}(n^2\kappa)$ bits per multiplication.

- Byzantine agreement (BA) is an MPC primitive which allows the players to agree on a particular value. It is used as a building block in many distributed protocols, and hence its efficiency is of particular importance. Known information-theoretically secure BA protocols with optimal resilience are very involved and inefficient – the most efficient known BA protocol requires $\mathcal{O}(n^{17}\kappa)$ bits of communication. We propose a new, conceptually simpler BA protocol communicating $\mathcal{O}(n^5\kappa)$ bits per BA.

- Lastly, we concentrate on MPC in asynchronous networks, where there is no upper bound on message delay. Known MPC protocols for the asynchronous setting suffer from two main disadvantages: they tend to have substantially higher communication complexity, and they do not allow to take the inputs of all honest players. We propose a solution to both these problems. We present a perfectly secure asynchronous MPC protocol that communicates only $\mathcal{O}(n^3\kappa)$ bits per multiplication. Furthermore, we extend the protocol for a hybrid communication model, allowing *all* players to give input if the *very first round* of the communication is synchronous, and taking at least $n - t$ inputs in a fully asynchronous setting.

All four mentioned contributions are optimal in all security parameters, i.e., achieve statistical security where $t < n/2$, and achieve perfect security where $t < n/3$ (respectively $t < n/4$ in the asynchronous world).

Zusammenfassung

Multi-Party Computation (MPC) ermöglicht einer Menge von n sich nicht vertrauenden Spielern, eine Funktion ihrer geheimen Inputs zu berechnen, so dass die Korrektheit des Outputs sowie die Privacy der Inputs garantiert ist, selbst wenn bis zu t der Spieler unehrlich sind und beliebig vom Protokoll abweichen.

In dieser Arbeit konzentrieren wir uns auf die *Effizienz* von MPC-Protokollen, mit Beiträgen in den folgenden Bereichen:

- Der grösste Kommunikationsaufwand in einer MPC wird für Multiplikationen erfordert. Seit kurzem können jedoch bei $t < n/3$ die Multiplikationen auf die nicht-robuste Generierung von Sharings von zufälligen Werten reduziert werden. Wir präsentieren eine neuartige Technik, die es erlaubt, perfekt sicher und sehr effizient solche zufälligen Sharings zu generieren. Basierend auf dieser Technik konstruieren wir ein perfekt sicheres MPC-Protokoll für $t < n/3$, mit einer Kommunikationkomplexität von $\mathcal{O}(n\kappa)$ Bits pro Multiplikation (wobei κ die Bit-Länge eines Wertes bezeichnet).

- Ein effizienter Ansatz zur Beschränkung der Anzahl Störungen, die unehrliche Spieler bei der Berechnung verursachen (und dadurch die Berechnung verlangsamen) können, ist Player-Elimination — jedesmal, wenn eine Inkonsistenz entdeckt wird, werden zwei Spieler lokalisiert und eliminiert, wobei mindestens einer der beiden Spielern unehrlich ist. Weil dabei auch ehrliche Spieler eliminiert werden können, ist diese Technik nur sehr begrenzt einsetzbar. Wir verallgemeinern und erweitern die Player-Elimination Technik, indem wir andere Methoden (als Elimination) finden, welche ebenfalls verhindern, dass lokalisierte Spieler die Berechnung

wiederholt stören können. Unsere neue Technik, genannt *Dispute-Control*, erlaubt die Konstruktion effizienter MPC-Protokolle in Modellen, in denen Player-Elimination nicht anwendbar ist: Wir präsentieren ein aktiv sicheres MPC-Protokoll mit optimaler Sicherheit (informations-theoretische Sicherheit bei einer unehrlichen Minderheit), welches nur $\mathcal{O}(n^2\kappa)$ Bits pro Multiplikation kommuniziert.

- Byzantine Agreement (BA) ist eine MPC-Primitive, mit welcher sich die Spieler auf einen Wert einigen können. Sie dient als Baustein in vielen verteilten Protokollen und ihre Effizienz ist deswegen von besonderer Bedeutung. Bekannte informations-theoretisch sichere BA-Protokolle mit optimaler Sicherheit sind sehr komplex und ineffizient – das effizienteste bekannte BA-Protokoll kommuniziert $\mathcal{O}(n^{17}\kappa)$ Bits. Wir schlagen ein neues, konzeptionell einfacheres BA-Protokoll vor, welches nur $\mathcal{O}(n^5\kappa)$ Bits kommuniziert.

- Schliesslich betrachten wir MPC in asynchronen Netzen, in denen Nachrichten im Netzwerk beliebig verzögert werden können. Bekannte MPC-Protokolle für asynchrone Netze leiden an zwei schwerwiegenden Nachteilen: Sie neigen zu hoher Kommunikationskomplexität und es können nicht die Inputs von allen ehrlichen Spielern berücksichtigt werden. Wir schlagen eine Lösung für diese beiden Probleme vor: Wir präsentieren ein perfekt sicheres asynchrones MPC-Protokoll, das nur $\mathcal{O}(n^3\kappa)$ Bits pro Multiplikation kommuniziert. Darüber hinaus erweitern wir das Protokoll für ein hybrides Modell, so dass *alle* Inputs berücksichtigt werden, falls die Nachrichten der *ersten Runde* des Protokolls synchron zugestellt werden, und mindestens $n-t$ Inputs bei voll asynchroner Kommunikation berücksichtigt werden.

Die vier genannten Beiträge sind optimal in allen Sicherheitsparametern, das bedeutet statistische Sicherheit für $t < n/2$, und perfekte Sicherheit für $t < n/3$ (bzw. $t < n/4$ im asynchronen Fall).

Contents

Chapter 1

Introduction

1.1 MPC

Secure multi-party computation (MPC) enables a set of mutually distrusting parties to perform some computation on their private inputs even when some of the parties misbehave and try to falsify the outcome of the computation or violate the privacy of the honest parties' inputs.

1.2 Models of MPC

We consider a set of parties consisting of a set of users $\mathcal{U}$, who can give input and get output and a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \dots, P_n\}$, who perform the computation. It is possible that a party plays both roles, the one of a user as well as the one of a player.

Many MPC protocols require the computation to be specified as an arithmetic circuit over a finite field $\mathbb{F}$ with input, addition, multiplication, random, and output gates. This kind of computation, in which all inputs can be given at the beginning of the computation is called *secure function evaluation (SFE)* or *non-reactive multi-party computation*.

Some protocols support the more general *reactive multi-party computation*, which allows to perform an arbitrary on-going (reactive) computation, where the users can give inputs and get outputs several times during the computation.

The faultiness of parties is modeled by a central adversary corrupting players and users.

1.2.1 Communication Model

We assume a complete network of secure (private and authentic) channels pair-wisely connecting all players as well as connecting each user to each player. According to timing assumptions, the following network models are considered:

Synchronous Networks In a synchronous network, there is a common global clock and the maximal delay of messages in the network is bounded by a known constant. Synchronous communication proceeds in rounds – all messages sent in one round are delivered at the beginning of the next round.

Asynchronous Networks In asynchronous networks, messages are delayed arbitrarily, in particular the order of the messages does not have to be preserved. However, every sent message will eventually be delivered. As a worst-case assumption, the adversary is given the power to schedule the delivery of messages. We model the privacy of the channels by considering oblivious schedulers – the only information known to the scheduler is the sender, the recipient and possibly the length of the message.

Hybrid Networks A hybrid network consist of few synchronous rounds followed by a fully asynchronous communication.

1.2.2 Adversary

We consider a central adversary corrupting parties (players and users). Once a party is corrupted it remains corrupted until the end of the computation.

The concrete adversary model is defined according to the following parameters:

Corruption Capabilities In this thesis we only consider threshold adversaries corrupting up to t of the players and any number of users.

Adversary Type A *passive (honest but curious) adversary* can read the internal state of the corrupted parties, trying to obtain some information he is not entitled to. An *active (malicious) adversary* can additionally make the corrupted parties deviate from the protocol in any desired manner, trying to falsify the outcome of the computation.

Adversary Power According to the computational power of the adversary, *unbounded* or *polynomially bounded* (in some security parameter) adversaries are considered.

Static vs. Adaptive A *Static* adversary has to choose which parties to corrupt before the start of the actual computation, whereas an *adaptive* adversary is allowed to corrupt the parties during the protocol execution (based on the information received so far).

Adversary in the Synchronous Model Remember that communication in the synchronous model proceeds in rounds and all messages sent in one round are delivered at the beginning of the next round. However in real life it is impossible to guarantee that all players send their messages simultaneously. To model the worst case scenario, in every round the adversary is given the right to read messages sent to corrupted players before sending out their messages (of the same round). Such an adversary is called *rushing*.

Adversary in the Asynchronous Model In the asynchronous communication model, the adversary can schedule the delivery of the messages in the network, i.e., he can delay any message arbitrarily. However, every sent message will eventually be delivered.

1.2.3 Security of MPC

Multi-party computation allows a set of parties to *securely* compute a function of their inputs, even if some players are corrupted by an adversary and are trying to disturb the computation.

Depending on the computational power of the tolerable adversary we speak about *computationally (or cryptographically)* secure protocols (tolerating a computationally-bounded adversary only) and *information-theoretically* secure protocols (tolerating an adversary with unlimited computational power). An information-theoretically secure protocol is called *perfectly* secure, if the computation is always secure, and *statistically* secure if some negligibly small error probability is allowed.

There are two main approaches in defining secure MPC: the property-based approach and the simulation-based approach.

In the property-based approach an MPC protocol is called *secure* if the following properties are satisfied:

- *Correctness:* The output of the computation is correct.
- *Privacy:* The adversary cannot obtain any information about the honest parties inputs (other than what he can compute from the corrupted parties' inputs and the outputs).
- *Robustness:* The adversary cannot abort the computation and prevent honest parties from receiving their output.

In case that robustness cannot be guaranteed, at least *fairness* is required: the adversary is not able to abort the computation after receiving some information about the output, i.e. the adversary cannot prevent the honest parties from receiving their output after receiving any information himself. A list of other desirable properties, as for example input independence, can be found in [MR98].

The more general and more precise simulation-based approach defines secure MPC with respect to a so-called specification. A *specification* is a protocol between the users and a trusted entity (trusted third party – TTP) performing the actual computation. The goal of MPC is the simulation of the trusted party. We say that a protocol *securely implements some specification* if the protocol achieves the same as the specification and the adversary cannot achieve more than what he could achieve in the specification. More precisely, the adversary can simulate the real world given the information he obtains in the ideal world (in the specification).

Precise simulation-based security definitions can be found in [Can98, Can00, Can01, BPW04].

The MPC protocols in this theses can be proven secure according to the security definition of [Can00].

Non-Robust (Detectable) Computation Whereas all our MPC protocols are robust, many of the used sub-protocols are not – the adversary can prevent the parties from getting their output. The idea of *non-robust (or detectable)* computation is the following: The adversary can cause the output to be incorrect, however this will be noticed by at least one honest player (who will inform other players and thus the output will not be used). More precisely: Every player has an internal state (the happy-bit), which is set to "happy" at the beginning of the computation. If a player detects a fault, he gets unhappy (sets his happy-bit to "unhappy'). We say that the protocol *succeeded* if all players remained happy, otherwise, the protocol *failed*.

Definition 1 *A* non-robust (or detectable) *protocol is a passively secure protocol with the following properties:*

- Completeness: *If all players follow the protocol then the protocol succeeds.*
- Correctness: *If the protocol succeeds than all outputs of the honest players are correct.*
- Privacy: *The privacy is guaranteed independently of the fact whether or not the protocol succeeds.*

1.3 History and Efficiency of MPC

The MPC problem dates back to Yao [Yao82]. The first generic solutions presented in [GMW87, CDG87, GHY87] (based on cryptographic intractability assumptions) and later in [BGW88, CCD88, RB89, Bea91b] (with information-theoretic security) assume the existence of a synchronous network.

Later [BCG93] initiated the study of MPC in the more realistic asynchronous networks, followed by [BKR94] (both with information-theoretic security).

1.3.1 Existential Results

The maximal number of players that can be tolerated to be corrupted depends on the type of the adversary (passive or active), on the underlying communication model (synchronous or asynchronous) and on whether or not a trusted setup is available.

Passive information-theoretically secure MPC is possible if and only if $t < n/2$ [BGW88, CCD88], whereas passive security against bounded adversaries is possible if and only if $t < n$ [GMW87].

Synchronous actively secure MPC from scratch (without trusted setup) is possible if and only if $t < n/3$. In this setting, the maximal possible security – perfect security – is achievable [BGW88]. If a trusted setup is available $t < n/2$ is possible. However, for $n/3 \leq t < n/2$, only statistical security can be achieved [RB89, Bea91b]. Thus, in both synchronous models (with and without setup), the maximal thresholds are achievable with information-theoretical security.

In the asynchronous setting, perfect information-theoretic security against an active adversary is possible if and only if $t < n/4$ [BCG93], whereas cryptographic and statistical security are possible if and only if $t < n/3$ [BT85, BKR94].

1.3.2 Efficiency of Synchronous MPC

In the last years, a lot of research concentrated on designing communication-efficient MPC protocols.

A number of works focused on designing protocols with only a few rounds of communication (e.g. [BB89, BMR90, BFKR90, DI05]), while others concentrated on minimizing the number of bits sent during the computation (e.g. [BFKR90, FY92, GRR98, CDD^{+}99, HMP00, CDN01, CDF01, HM01, HN05, HN06, DN07]). The cryptographically secure protocol of [DI06] combines low round complexity with a low bit-complexity, however for the price of a non-optimal threshold.

The main efficiency bottleneck of bit-efficient MPC protocols is the computation of the multiplication gates. The following table gives an overview on the currently (excluding results presented in this thesis) most bit-efficient synchronous MPC protocols (with an optimal threshold in the respective security model). Where κ denotes the bit-length of a field element (resp. the security parameter), D the multiplicative depth of the circuit, and $\mathcal{BC}(\cdot)$ the number of broadcasted bits.

Security Model	**Thresh.**	**Bits/Mult.**	**Reference**
perfect passive	$t < n/2$	$\mathcal{O}(n\kappa)$	[DN07]
cryptographic	$t < n/2$	$\mathcal{O}(n\kappa)$	[HN06]
statistical	$t < n/3$	$\mathcal{O}(n\kappa + n^2 D)$	[DN07]
perfect	$t < n/3$	$\mathcal{O}(n^3\kappa)$	[HMP00]
statist. with setup	$t < n/2$	$\mathcal{O}(n^5)\,\mathcal{BA}(\kappa)$	[CDD$^+$99]

1.3.3 Efficiency of Synchronous Byzantine Agreement

Byzantine agreement enables the parties to reach agreement on some value.[1]

In the synchronous model without trusted setup, Byzantine agreement among n players is achievable for $t < n/3$ communicating $\mathcal{O}(n^2)$ bits [BGP92, CW92]. In the model with a trusted setup, the communication complexity of BA heavily depends on whether information-theoretic security is required or cryptographic security is sufficient. When cryptographic security is sufficient, then $\mathcal{O}(n^3\kappa)$ bits are sufficient for reaching agreement, where κ denotes the security parameter [DS83]. When information-theoretic security is desired, then reaching agreement with the currently most efficient BA protocols requires at least $\mathcal{O}(n^6\kappa)$ bits of communication [BPW91, PW96, Fit03].

However, the latter result *consumes* the setup, i.e., a given setup can be used only for one single BA operation. Of course, one can start with an m times larger setup which supports m BA operations, but the number of BA operations is a priori fixed, and the size of the setup grows linearly with the number of intended BA operations. This diametrically contrasts the cryptographic scenario, where a fixed-size setup is sufficient for polynomially many BA operations. In [PW96], a method for *refreshing* the setup is shown: They start with a compact setup, use some part of the setup to perform the effective BA operation, and the remaining setup to generate a new, full-fledged setup. With this approach, a constant-size setup is sufficient for polynomially many BA invocations. However, with every BA invocation, the setup must be refreshed, which requires a communication of $\mathcal{O}(n^{17}\kappa)$ bits [PW96, Fit03]. Hence, when the initial setup should be compact, then the costs for a BA operation of [PW96] is as high as $\mathcal{O}(n^{17}\kappa)$ bits.

[1]For more precise definition see Section 2.3.

1.3.4 Efficiency of Asynchronous MPC

The first MPC protocols for asynchronous networks feature (impractically) high communication complexities. The most efficient asynchronous protocol is the one of [HNP08] communicating $\mathcal{O}(n^2)$ per multiplication while providing cryptographic security only. The most efficient information-theoretically secure protocols up to now were proposed in [SR00, PSR02]. Both protocols are secure against an unbounded adversary corrupting up to $t < n/4$ players. The first one makes extensive use of the (communication-intensive) BA primitive – $\mathcal{O}(n^2)$ invocations per multiplication, which amounts to $\Omega(n^5)$ bits of communication per multiplication.[2] The second one requires only $\mathcal{O}(n^2)$ invocations to BA in total, however, still communicates $\mathcal{O}(n^4)$ bits per multiplication, and provides statistical security only (for which $t < n/4$ is not optimal).

1.4 Contributions

In this theses we concentrate on bit-complexity of MPC protocols, measured in bits sent by honest parties.

In the synchronous model, we consider the two settings optimal in all security parameters:

- MPC from scratch: with perfect security and $t < n/3$
- MPC with setup: with statistical security and $t < n/2$.

In the asynchronous model, only the perfect $t < n/4$ setting is considered.

Additionally in the synchronous setting with setup, we propose an efficient protocol for statistically secure Byzantine agreement with $t < n/2$.

1.4.1 Actively Secure MPC from Scratch

The main efficiency bottleneck in MPC is the computation of the multiplication gates. However, thanks to techniques from [Bea91a, HMP00,

[2]The most efficient known asynchronous BA protocol requires $\Omega(n^3)$.

DN07], for $t < n/3$, multiplication can be reduced to non-robust generation of correct sharings of secret random values. Up to now, the most efficient way to generate random values was the following: First every player privately shares a random value, and the correctness of these sharings is checked by checking a blinded random linear combination of the sharings. Then, if the check succeeds, a set of independent secret random sharings is extracted from the original sharings using super-invertible matrices. While being very efficient, this approach has two drawbacks: it requires the generation of a random challenge (which again requires generation of random sharings), and it yield an error probability.

We present a novel technique which, at the same time, allows to perfectly and very efficiently verify a bunch of sharings and (if successful) to extract a set of (new) correct random sharings given that a sub-set of the original sharings is random.

More precisely, given n supposedly random sharings, up to t of them distributed by corrupted players (and thus possibly inconsistent, non-random, etc), we can check whether they are all correct, and if so, (locally) compute $n - 2t$ correct and uniform random sharings. The check is (despite of being perfectly secure) highly efficient; it only requires the reconstruction of $2t$ sharings, each towards a single player.

In other words, we can non-robustly but detectably generate $\Omega(n)$ uniform random sharings, unknown to the adversary, with perfect security and communicating $\mathcal{O}(n^2)$ field elements.

The novel technique is based on so-called *hyper-invertible matrices*, i.e., matrices whose every square sub-matrix is invertible. Applying n sharings to such a matrix results in n sharings with the property that (i) if *any* (up to t) of the inputs sharings are inconsistent, then this can be seen in *every* subset of t output sharings, and (ii) if *any* $n - t$ input sharings are uniform random, then *every* subset of size $n - t$ of output sharings is uniform random.

Using hyper-invertible matrices and some techniques from [Bea91a, HMP00, DN07], we construct a perfectly secure multi-party protocol with optimal resilience and linear communication complexity. This can be seen as an efficiency improvement (the most efficient known MPC protocol with perfect security communicates $\mathcal{O}(n^3)$ field elements per multiplication [HMP00]), or alternatively as a security improvement (the most secure known MPC protocol with linear communication provides error probability [DN07]). In either case, we consider the new protocol to be more elegant, as it employs neither two-dimensional sharings (like all previous perfectly-secure MPC protocols) nor probabilistic checks (like all previous MPC protocols with linear communication complexity).

These results were presented in [BH08]

1.4.2 Actively Secure MPC with Setup

In this work, we show that information-theoretic MPC with adaptive active security for $t < n/2$ is achievable with sending $\mathcal{O}(n^2)$ field elements per multiplication, and broadcasting $\mathcal{O}(n^3)$ field elements *overall*, for the whole computation. This improves on previous protocols for this setting which require *broadcasting* $\Omega(n^5)$ field elements per multiplication [CDD+99].

At the time of the publication of this result, all known MPC protocols required at least $\mathcal{O}(n^2\kappa)$ bits of communication per multiplication, even protocols providing security against passive or bounded adversaries only.

Technically, the new protocol improves the approach of [CDD+99], which requires $\Omega(n^5)$ broadcasts per multiplication. We introduce a new concept, so-called *dispute control*, that allows to substantially reduce the communication complexity. The goal of dispute control is to reduce the frequency of faults that the adversary can provoke by identifying a pair of disputing players (at least one of them corrupted) whenever a fault is observed and preventing this pair from getting into dispute ever again. Hence, the number of faults that can occur during the whole protocol is limited to $t(t+1)$. This technique is inspired by the player-elimination framework [HMP00], and shares many advantages with it. However, player elimination is not to be suited for models with $t \geq n/3$.

These results were presented in [BH06]

1.4.3 Broadcast

We present a protocol for information-theoretically secure Byzantine agreement (both consensus and broadcast) which communicates $\mathcal{O}(n^4\kappa)$ bits when the setup may be consumed (i.e., the number of BA operations per setup is a priori fixed). This contrasts to the communication complexity of $\mathcal{O}(n^6\kappa)$ bits of previous information-theoretically secure BA protocols [BPW91, PW96].

More importantly, we present a refresh operation for our BA protocol, communicating only $\mathcal{O}(n^5\kappa)$ bits, contrasting the complexity of $\mathcal{O}(n^{17}\kappa)$ bits of previous refresh protocols [PW96]. This new result allows for polynomially many information-theoretically secure BA operations from a fixed-size setup, where each BA operations costs $\mathcal{O}(n^5\kappa)$ bits.

This substantial speed-up is primarily due to a new concept, namely that the refresh operation does not need to succeed all the time. Whenever the setup is to be refreshed, the players try to do so, but if they fail, they pick a fresh setup from an a priori prepared stock. Furthermore, using techniques from the player-elimination framework [HMP00], the number of failed refresh operations can be limited to t. Using algebraic information-theoretic pseudo-signatures [SHZI02] for appropriate parameters, the function to be computed in the refresh protocol becomes algebraic, more precisely a circuit over a finite field with multiplicative depth 1. Such a function is very well suited for efficient non-robust computation; in fact, it can be computed based on a simple one-dimensional Shamir-sharing, although $t < n/2$.[3] This allows a very simple refresh protocol with low communication overhead.

Compared to the refresh protocol of [PW96], our refresh protocol has the disadvantage that it requires $t < n/2$, whereas the protocol of [PW96] can cope with $t < n$. However, almost all applications using BA as subprotocol (like voting, biding, multi-party computation, etc.) inherently require $t < n/2$, hence the limitation on our BA protocol is usually of theoretical relevance only.

These results were published in [BHR07].

1.4.4 Asynchronous MPC

Known MPC protocols for the asynchronous setting suffer from two main disadvantages in contrast to their more restrictive synchronous counterparts, both significantly reducing their practicability: Asynchronous protocol tend to have substantially higher communication complexity, and they do not allow to take the inputs of all honest players. We propose a solution to both these problems.

First, we present a perfectly secure asynchronous MPC protocol that communicates only $\mathcal{O}(n^3)$ field elements per multiplication. At the time

[3]Note that general MPC protocols for this model need a three-level sharing, namely a two-dimensional Shamir sharing ameliorated with authentication tags [RB89, Bea91b, BH06].

of the publication of this result, the same communication complexity was also required by the most efficient known perfectly secure protocol for the synchronous model [HMP00], as well as by the most efficient asynchronous protocol only secure against computationally bounded adversaries [HNP05]. The protocol provides perfect security against an unbounded adaptive active adversary corrupting up to $t < n/4$ players, which is optimal. In contrast to the previous asynchronous protocols, the new protocol is very simple.

Second, we extended the protocol for a hybrid communication model (with the same security properties and the same communication complexity), allowing *all* players to give input if the *very first round* of the communication is synchronous, and takes at least $n - t$ inputs in a fully asynchronous setting. It is well-known that fully asynchronous protocols cannot take the inputs of all players; however, we show that a single round of synchronous communication is sufficient to take all inputs. We stress that it is important that this round is the first round, because assuming the k-th round to be synchronous implies that all rounds up to k must also be synchronous. Furthermore, the protocol achieves the best of both worlds, i.e., takes the inputs of *all* players when indeed the first round is synchronous, and still takes the inputs of at least $n - t$ players even if the synchrony assumptions cannot be fulfilled. More precisely, the protocol takes the inputs of at least $n - t$ players, and additionally, always takes the inputs of players whose first-round messages are delivered synchronously.

These results were published in [BH07].

Chapter 2

Preliminaries

2.1 Generic Approach to MPC

The goal of MPC is to allow a set of parties to perform some computation on their inputs, such that the privacy of the inputs as well as the correctness of the outputs is guaranteed, even in the presence of an adversary corrupting some players.

The main tool for designing protocols secure against passive (honest but curious) adversaries is *secret sharing* [Bla79, Sha79]. Secret sharing enables the parties to divide (share) their input (or any other secret) among the players such that any small-enough set of players has no joint information about the secret, but any large-enough set of players has enough information to uniquely determine (reconstruct) the secret.

Using secret sharing, the computation proceeds as follows: First, all users share their inputs among the players. Then the circuit is evaluated gate by gate on the shared values, such that every intermediate value is shared among the players. At the end, the output is reconstructed.

Thus, most MPC protocols consist of sub-protocols for sharing and reconstructing values, and of sub-protocols for adding (multiplying) shared values, such that the sum (product) is shared among the players and no information on the summands (factors) and the sum (product) leaks to the adversary.

Actively secure protocols employ zero-knowledge proofs and consistency checks to prevent faults caused by misbehaving players.

In the following two section we describe the two most important primitives of information-theoretically secure MPC: Secret Sharing and Byzantine Agreement.

2.2 Secret Sharing

The intuition of secret sharing is to enable a dealer D to divide (share) a secret s among the players (by sending each player P_i a so called share s_i), such that the corrupted players have no joint information about the secret, however all players together can (later) reconstruct the secret.

Definition 2 *A* Verifiable Secret Sharing Scheme (VSS) *(for a dealer holding a secret s) is a pair of protocols* VShare *and* VRec *with the following properties:*

- *Termination: Once the protocol* VShare *(*VRec*) is invoked, every honest player will complete it.*
- *Correctness: After termination of* VShare *there exists a fixed value $r \in \mathbb{F}$ such that:*
  - *If the dealer was honest during* VShare*, than r is his secret, i.e. $r = s$.*
  - *Every honest player outputs r upon completing* VRec*.*
- *Privacy: If the dealer is honest, then the adversary obtains no information about the shared secret before* VRec *is invoked.*

Such VSS is achievable in the model with synchronous channels only. In the asynchronous setting only a weaker termination property can be achieved.

Definition 3 *An* Asynchronous Verifiable Secret Sharing Scheme (AVSS) *(for a dealer holding a secret s) is a pair of protocols* VShare *and* VRec *with the following properties:*

- *Termination:*
  - *If the dealer is honest, than every honest player will eventually complete* VShare*.*

  - *If one honest player completes* VShare*, then all honest players will eventually complete* VShare*.*
  - *If one honest player completes* VShare*, then all honest players will eventually complete* VRec *(once it is invoked).*

- *Correctness: After one honest player has completed* VShare*, there exists a fixed value $r \in \mathbb{F}$ such that:*
  - *If the dealer was honest during* VShare*, than r is his secret, i.e. $r = s$.*
  - *Every honest player outputs r upon completing* VRec*.*
- *Privacy: If the dealer is honest, then the adversary obtains no information about the shared secret before* VRec *is invoked by at least one honest player.*

These definitions themselves do not guarantee the feasibility of any computation on the shared values and are thus not sufficient in the context of multi-party computation.

In this thesis we will describe a VSS scheme by defining a *correct sharing* of a value (a state uniquely defining the shared value) and by describing the protocol VShare which produces a correct sharing (of the honest dealers secret), and the protocol VRec, which given a correct sharing reconstructs the shared secret.

All our sharings are based on the Shamir sharing scheme [Sha79]. To every player P_i a unique fixed value $\alpha_i \in \mathbb{F} \setminus \{0\}$ is publicly assigned and the shares are evaluations of a polynomial at the points $\alpha_1, \ldots, \alpha_n$.

2.3 Consistency Primitives

An active adversary can disturb the computation by trying to cause inconsistencies in the views of the honest players. For example if a player is supposed to send one value to all players, an actively corrupted player might sent different values to different players. To cope with this problem some consistency primitives are needed.

The problem of Byzantine agreement (BA), as originally proposed by Pease, Shostak, and Lamport [PSL80, LSP82], is the following: n players $P_1, \ldots, P_n$ want to reach agreement on some value v, but up to t of them are faulty and try to prevent the others from reaching agreement. There are two flavors of the BA problem: In the broadcast problem, a designated player (the sender) holds an input message m, and all players should learn m and agree on it. In the consensus problem, every player P_i holds (supposedly the same) message m_i, and the players want to agree on this message.

More formally, a protocol with P_S giving input m is a *broadcast protocol*, when every honest P_i outputs the same message $m'_i = m'$ for some m' (*consistency*), and when $m' = m$, given that P_S is honest (*validity*). Analogously, a protocol with every player P_i giving input m_i is a *consensus protocol*, when every honest P_i outputs $m'_i = m'$ for some m' (*consistency*), and when $m' = m$, given that every honest P_i inputs the same message $m_i = m$ for some m (*validity*).

The feasibility of broadcast and consensus depends on whether or not a trusted setup (e.g. a PKI setup) is available. When no trusted setup is available ("from scratch"), then consensus and broadcast are achievable if and only if at most $t < n/3$ players are corrupted. When a trusted setup is available, then consensus is achievable if and only if at most $t < n/2$ players are corrupted, and broadcast is achievable if and only if at most $t < n$ players are corrupted. All bounds can be achieved with information-theoretical security, and the bounds are tight even with respect to cryptographic security. We stress in particular that no broadcast protocol (even with cryptographic intractability assumptions) can exceed the $t < n/3$ bound unless it can rely on a trusted setup [FLM86, Fit03]. The main difference between protocols with information-theoretic security and those with cryptographic security is their efficiency.

Note that in asynchronous networks, slightly different definitions of BA are used (and will be explained in Chapter 7).

Chapter 3

Passive MPC

3.1 Introduction

In this chapter we present an MPC protocol perfectly secure against a passive adversary corrupting up to $t < n/2$ players and any number of users. Remember that passively corrupted players correctly follow the protocol, however the adversary has a complete view on their internal state.

The first MPC protocol perfectly secure against a passive adversary corrupting up to $t < n/2$ players was the one of [BGW88]; communicating $\mathcal{O}(n^2\kappa)$ bits per multiplication. Recently [DN07] presented a protocol for the same model, communicating only $\mathcal{O}(n\kappa)$ bits per multiplication. Both protocols use the standard Shamir-sharing scheme [Sha79] (and the same share and reconstruct protocols), however [DN07] provides more efficient sub-protocols for generating random values and for multiplication.

In the following sections we explain the formal model, define a correct secret sharing and present the protocols for sharing and reconstructing values. Then we explain how to perform computations on the shared values, i.e. how to add shared values, how to generate sharings of random values and how to multiply shared values.

This passively secure (sub-)protocols provide a basis for the actively secure (sub-)protocols of the following chapters. As parts of our work are based on the (simpler) approach of [BGW88] (with some simplifications from [GRR98]) whereas other parts follow the (more efficient) approach of [DN07], we will present both.

3.2 Model

We consider a set $\mathcal{U}$ of users, who can give input and receive output, and a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, who perform the computation. The players and users are connected by a complete network of secure (private and authentic) channels.

The function to be computed is specified as an arithmetic circuit over a finite field $\mathbb{F}$ (with $|\mathbb{F}| > n$), with input, addition, multiplication, random, and output gates. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O, respectively. We use κ to denote the bit-length of the elements in $\mathbb{F}$, i.e. $\kappa = log|\mathbb{F}|$.

The faultiness of players or users is modeled in terms of a central adversary corrupting players and users. The adversary can corrupt up to $t < n/2$ players and any number of users. The adversary is computationally unbounded, passive, and adaptive. The security of the protocol is perfect, i.e., information-theoretic without any error probability.

To every player $P_i \in \mathcal{P}$ a unique, non-zero element $\alpha_i \in \mathbb{F} \setminus \{0\}$ is assigned.

3.3 Secret Sharing

In this section we first define a correct sharing of a value (and introduce some notation) and then present protocols for sharing values and reconstructing sharings.

Remember that as we are dealing with a passive adversary only, our sole concern is privacy.

3.3.1 Definitions and Notations

As secret-sharing scheme, we use the standard Shamir sharing scheme [Sha79].

Definition 4 *A value s is (correctly) d-*shared *(among the players in $\mathcal{P}$) if every player $P_i \in \mathcal{P}$ is holding a share s_i of s, such that there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$ and $p(\alpha_i) = s_i$ for every $P_i \in \mathcal{P}$.*[4] *The vector $(s_1, \ldots, s_n)$ of shares is called a d-sharing of s, and is denoted by $[s]_d$. A (possibly incomplete) set of shares is called d-*consistent *if these shares lie on a degree d polynomial.*

Note that a d-sharing $[s]_d = (s_1, \ldots, s_n)$ well-defines a value if and only if $d < n$ and a random d-sharing does not leak any information to the adversary if and only if $d \geq t$. Thus all sharings used in our protocols will be d-sharings with $t \leq d < n$.

By saying that the players in $\mathcal{P}$ compute (locally)

$$([y^{(1)}]_{d'}, \ldots, [y^{(m')}]_{d'}) = f([x^{(1)}]_d, \ldots, [x^{(m)}]_d)$$

(for any function $f : \mathbb{F}^m \rightarrow \mathbb{F}^{m'}$) we mean that every player P_i applies this function to his shares, i.e. computes

$$(y_i^{(1)}, \ldots, y_i^{(m')}) = f(x_i^{(1)}, \ldots, x_i^{(m)}).$$

Note that by applying any linear or affine function to correct d-sharings we get a correct d-sharing of the output. However, by multiplying two correct d-sharings we get a correct $2d$-sharing of the product, i.e. $[a]_d[b]_d = [ab]_{2d}$.

At some point we will use so called *double-sharings*.

Definition 5 *A value x is (d, d')-*shared *among the players $\mathcal{P}$, denoted as $[x]_{d,d'}$, if x is both d-shared and d'-shared. We denote such a sharing as a* double-sharing, *and the pair of shares held by each player as his* double-share.

We (trivially) observe that any sum of correct (d, d')-sharings is a correct (d, d')-sharing of the sum.

3.3.2 The Share Protocol

The following protocol allows a dealer (a player or a user) to privately share his secret s. The dealer P_D chooses a random degree-d polynomial $p(\cdot)$ with $p(0) = s$ and distributes the shares among the players in $\mathcal{P}$.

[4]Where α_i denotes the unique fixed value assigned to P_i.

Protocol $\mathsf{Share}(P_D \in (\mathcal{P} \cup \mathcal{U}), s, d)$**.**

1. DISTRIBUTE SHARES: P_D chooses a random degree-d polynomial $p(\cdot)$ with $s = p(0)$ and sends to every player $P_i \in \mathcal{P}$ his share $s_i = p(\alpha_i)$.
2. OUTPUT: Every player P_i outputs the received share s_i.

The sharing $(s_1, \dots, s_n)$, defined by the shares outputted by the players $P_1, \dots, P_n$, is a correct d-sharing of s, i.e. $(s_1, \dots, s_n) = [s]_d$. The protocol Share (and the sharing $[s]_d$ itself) leaks no information about the shared secret as long as $d \geq t$. Share communicates $n\kappa$ bits.

3.3.3 The Reconstruct Protocol

We present two reconstruction protocols, one for private reconstruction (towards a designated recipient P_R) and one for public reconstruction (towards every player in $\mathcal{P}$).

Technically, public reconstruction can be achieved by n private reconstructions – one to each player in $\mathcal{P}$. However, the communication costs of such public reconstruction are n times the costs of private reconstruction, which is unnecessarily high.

To privately reconstruct a shared secret s towards a designated recipient P_R, every player sends his share of $[s]_d$ to the recipient. P_R then reconstructs the secret, by finding the (unique) degree-d polynomial $p(\cdot)$ with $p(\alpha_i) = s_i$ for every $i = 1, \dots, n$, using the Lagrange interpolation formula

$$p(x) = \sum_i \prod_{j \neq i} \frac{x - \alpha_j}{\alpha_i - \alpha_j} s_i$$

and computing s as $s = p(0)$. Respectively the secret can be computed directly by setting $x = 0$ in the above formula

$$s = p(0) = \sum_i w_i s_i \text{ with } w_i = \prod_{j \neq i} \frac{\alpha_j}{\alpha_j - \alpha_i}.$$

Protocol $\mathsf{ReconsPrivPassive}(P_R \in (\mathcal{P} \cup \mathcal{U}), d, [s]_d)$**.**

1. SEND SHARES: Every player $P_i \in \mathcal{P}$ sends his share s_i of s to P_R.
2. INTERPOLATE: P_R interpolates the received shares, i.e. he computes s as $s = \sum_i w_i s_i$ with $w_i = \prod_{j \neq i} \frac{\alpha_j}{\alpha_j - \alpha_i}$.

3. OUTPUT: P_R outputs s.

The protocol ReconsPrivPassive privately reconstructs every correctly d-shared secret for $d < n$, communicating $n\kappa$ bits. If the recipient is honest, the adversary obtains no information on s.

For public reconstruction (in $\mathcal{P}$) the sharing $[s]_d$ is first reconstructed towards one designated player (say P_1), who then distributes the reconstructed value s among the other players in $\mathcal{P}$.

Protocol ReconsPublicPassive(d, $[s]_d$)**.**

1. RECONSTRUCT TOWARDS P_1: ReconsPrivPassive($P_1, d, [s]_d$) is invoked to reconstruct s towards P_1.
2. DISTRIBUTE: P_1 sends the reconstructed secret s to every $P_i \in \mathcal{P}$.
3. OUTPUT: Every player outputs the received s.

The protocol ReconsPublicPassive publicly reconstructs every correctly d-shared secret for $d < n$, communicating $2n\kappa$ bits.

3.4 Computing Affine Functions

As already mentioned, for any affine function $f : \mathbb{F}^m \to \mathbb{F}$ and any correct d-sharings $[x^{(1)}]_d, \ldots, [x^{(m)}]_d$ it holds that $[x]_d$ defined as

$$[x]_d = f([x^{(1)}]_d, \ldots, [x^{(m)}]_d)$$

is a correct d-sharing of $x = f(x^{(1)}, \ldots, x^{(m)})$.

This means, that any affine function can be computed locally without any communication, by every player applying the function to his respective shares.

3.5 Generating Random Sharings

In this section we present a protocol for generating secret, uniformly-random values shared among the players.

We first present a trivial protocol which generates one random sharing with communication costs $\mathcal{O}(n^2\kappa)$ bits and then a more involved protocol (along the lines of [HN06]) which has the same overall complexity while generating $\mathcal{O}(n)$ independent random sharings.

Both protocols use the fact, that by summing up a bunch of values out of which at least one is secret, random and independent of the others, we get a secret random value.

3.5.1 Simple Approach

The following protocol ShareRandomPassiveSimple(d) generates a d-sharing of a secret random value as a sum of n shared random contributions (one from each player).

Protocol ShareRandomPassiveSimple(d)**.**

1. SHARE: Every player $P_i \in \mathcal{P}$ chooses a uniform random value $r_i \in_R \mathbb{F}$ and d-shares it, acting as a dealer in Share (P_i, r_i, d) – resulting in $[r_i]_d$.
2. OUTPUT: The players output the d-sharing $[r]_d = \sum_{i=1}^{n} [r_i]_d$ of the sum $r = \sum_{i=1}^{n} r_i$.

As there is at least one honest player who contributed a uniform random value unknown to the adversary in Step 1, the value generated by ShareRandomPassiveSimple is uniformly random and unknown to the adversary. The protocol ShareRandomPassiveSimple communicates $n^2\kappa$ bits.

As one secret random contribution is sufficient to make the sum secret and random, the complexity of ShareRandomPassiveSimple could be reduced a little by having only $t+1$ players share a random value in Step 1 (instead of all n). However such protocol would still communicate $\mathcal{O}(n^2\kappa)$ bits.

In the next section we present super-invertible matrices, which enable us to exploit the maximum: generate $n-t$ secret random sharings from n contributions out of which $n-t$ come from honest players (and thus are random and unknown to the adversary).

3.5.2 Efficient Approach

In the following we first introduce super-invertible matrices (SIM) and then show how to use them to efficiently generate random sharings. This technique was introduced by [HN06].

We consider r-by-c matrices M over the field $\mathbb{F}$. When $r = c$, M is called *invertible* if all column-vectors are linearly independent. When $r \leq c$, M is called *super-invertible* if every subset of r column-vectors are linearly independent. Formally, for an r-by-c matrix M and an index set $C \subseteq \{1, \ldots, c\}$, we denote by M_C the matrix consisting of the columns $i \in C$ of M. Then, M is super-invertible if for all C with $|C| = r$, M_C is invertible.

Super-invertible matrices are of great help to extract random elements from a set of some random and some non-random elements: Consider a vector $(x_1, \ldots, x_c)$ of elements, where for some $C \subseteq \{1, \ldots, c\}$ with $|C| = r$, the elements $\{x_i\}_{i \in C}$ are chosen uniformly at random (by honest players), and the elements $\{x_j\}_{j \notin C}$ are chosen maliciously (by corrupted players). Then, the vector $(y_1, \ldots, y_r) = M(x_1, \ldots, x_c)$ is uniformly random and unknown to the adversary.[5]

In the following protocol every player chooses and shares a random value, resulting in n shared values out of which at least $n - t$ are random and unknown to the adversary. Then using an $n \times n - t$ super-invertible matrix M $n - t$ sharings of random and secret values are computed (locally).

Protocol ShareRandomPassive(d)**.**

1. SHARE: Every player $P_i \in \mathcal{P}$ chooses a uniform random value $s_i \in_R \mathbb{F}$ and d-shares it – resulting in $[s_i]_d$.
2. APPLY M: The players (locally) compute
$$\big([r_1]_d, \ldots, [r_{n-t}]_d\big) = M\big([s_1]_d, \ldots, [s_n]_d\big).$$
In order to do so, every player computes his share of each r_j as the respective linear combination of his shares of the s_i-values.
3. OUTPUT: The $n - t$ sharings $[r_1]_d, \ldots, [r_{n-t}]_d$ are outputted.

ShareRandomPassive generates $n - t > n/2$ secret random sharings while communicating $n^2\kappa$ bits. Thus the amortized communication complexity of ShareRandomPassive is less than $2n\kappa$ bits per random sharing.

[5]This follows from the observation that the $c - r$ maliciously chosen elements $\{x_j\}_{j \notin C}$ define a bijection from the r random elements $\{x_i\}_{i \in C}$ onto $(y_1, \ldots, y_r)$.

3.6 Multiplication

The goal of this section is to present a protocol for computing a d-sharing of the product of two d-shared values x and y.

Remember that $[x]_d[y]_d = [xy]_{2d}$, thus by having every player multiply his shares of x and y, a correct $2d$-sharing of the product xy can be computed without any communication. What remains to be done is to reduce this $2d$-sharing to an independent d-sharing of xy (without revealing any information to the adversary). This can be done for $d < n/2$ (for $d \geq n/2$ a $2d$-sharing does not well-define any value).

We first present a simpler multiplication protocol from [GRR98] with quadratic communication complexity and then the more efficient multiplication protocol from [DN07] with linear communication complexity.

3.6.1 Simple Approach

Remember that a d-shared value s (with $d < n$) can be computed as a linear combination of the shares $s_1, \dots s_n$ (using the Lagrange interpolation formula):

$$s = \sum_{i=1}^{n} w_i s_i \;\text{ with }\; w_i = \prod_{j \neq i} \frac{\alpha_i}{\alpha_j - \alpha_i}.$$

The same holds for any d'-sharings of s and $s_1, \dots, s_n$: given the n d'-sharings $[s_1]_{d'}, \dots, [s_n]_{d'}$, the d'-sharing of s can be computed as:

$$[s]_{d'} = \sum_{i=1}^{n} w_i [s_i]_{d'} \;\text{ with }\; w_i = \prod_{j \neq i} \frac{\alpha_i}{\alpha_j - \alpha_i}.$$

This is used in the following protocol MultPassiveSimple (from [GRR98]). The d-sharing $[z]_d$ of the product $z0xy$ is computed from the (locally computed) $2d$-sharing $[z]_{2d} = [xy]_{2d} = [x]_d[y]_d$ by re-sharing: every player d-shares his $2d$-share of $[z]_{2d}$ and the d-sharing $[z]_d$ is computed as a linear combination of the share-sharings (according to Lagrange interpolation).

Protocol MultPassiveSimple($[x]_d, [y]_d$).

1. MULTIPLY SHARES: The players compute (locally) the $2d$-sharing $[z]_{2d}$ of $z = xy$ as $[z]_{2d} = [x]_d[y]_d$ (by every player computing the product of his shares).
2. REDUCE DEGREE:
   - 2.1 RE-SHARE: Every player P_i d-shares his $2d$-share z_i of z, acting as a dealer in Share (P_i, z_i, d) – resulting in $[z_i]_d$.
   - 2.1 INTERPOLATE: The players compute (locally) the d-sharing $[z]_d = \sum_{i=1}^{n} w_i[z_i]_d$ with $w_i = \prod_{j \neq i} \frac{\alpha_i}{\alpha_j - \alpha_i}$.
3. OUTPUT: The players output the d-sharing $[z]_d$.

The above protocol correctly and privately multiplies any two d-shared values for $2d < n$. The communication complexity of MultPassiveSimple is n times the communication complexity of Share, i.e. $n^2\kappa$ bits. In the following we present a more efficient multiplication protocol.

3.6.2 Efficient Approach

We first present the protocol MultPassive (from [DN07]) which efficiently multiplies two d-shared values x and y, given a secret random value r which is both d- and $2d$-shared. Then we show how many such secret random double-sharings can be efficiently generated.

The main idea of MultPassive is to reduce $[xy]_{2d}$ to $[xy]_d$ by publicly reconstructing the difference $[\delta]_{2d} = [xy]_{2d} - [r]_{2d}$ and then (locally) computing $[xy]_d$ as $[xy]_d = [r]_d + \delta$.

Protocol MultPassive($[x]_d, [y]_d, [r]_{d,2d}$).

1. MULTIPLY SHARES: The players compute (locally) the $2d$-sharing $[z]_{2d}$ of $z = xy$ as $[z]_{2d} = [x]_d[y]_d$ (by every player computing the product of his shares).
2. REDUCE DEGREE
   - 2.1 The players compute (locally) a $2d$-sharing of the difference $\delta = z - r$ by computing $[\delta]_{2d} = [z]_{2d} - [r]_{2d}$.
   - 2.2 Invoke ReconsPublicPassive($[\delta]_{2d}$) to reconstruct the difference δ towards every player in $\mathcal{P}$.
   - 2.3 The players compute (locally) the d-sharing $[z]_d = [r]_d + \delta$.

3. OUTPUT: The players output the d-sharing $[z]_d$.

The protocol MultPassive correctly multiplies any two d-shared values for $2d < n$. The protocol MultPassive is private given that r is a secret random value (as then reconstructing $\delta = z - r$ leaks no information on z). The communication complexity of MultPassive is the same as the one of ReconsPublicPassive, i.e. $2n\kappa$ bits.

The following protocol DoubleShareRandomPassive(d, d') for efficient generation of many secret random double-sharings works along the lines of the protocol ShareRandomPassive(d) using a $n \times n - t$ super-invertible matrix M.

Protocol DoubleShareRandomPassive(d, d')**.**

1. SHARE: Every player $P_i \in \mathcal{P}$ chooses a uniform random value $s_i \in_R \mathbb{F}$ and shares it once with degree d and once with degree d' acting as a dealer in Share(P_i, s_i, d) and Share(P_i, s_i, d') – resulting in a double-sharing $[s_i]_{d,d'}$.
2. APPLY M: The players (locally) compute

$$\big([r_1]_{d,d'}, \ldots, [r_{n-t}]_{d,d'}\big) = M\big([s_1]_{d,d'}, \ldots, [s_n]_{d,d'}\big).$$

   In order to do so, every player computes his double-share of each r_j as the respective linear combination of his double-shares of the s_i-values.
3. OUTPUT: The $n - t$ double-sharings $[r_1]_{d,d'}, \ldots, [r_{n-t}]_{d,d'}$ are outputted.

The protocol DoubleShareRandomPassive generates independent random double-sharings of $n - t$ independent secret random values communicating $2n^2\kappa$ bits. Thus the amortized communication complexity of DoubleShareRandomPassive is less than $4n\kappa$ bits per one random double-sharing.

3.7 The Passively-Secure MPC Protocol

The protocol PassiveMPC (from [DN07]) proceeds in two phases: the preparation phase and the computation phase.

In the preparation phase many $(t, 2t)$-sharings of secret random values are generated (in parallel), one for every multiplication gate. Furthermore, for every random gate a t-sharing of a random r is generated. For the sake of simplicity, we generate $c_M + c_R$ random $(t, 2t)$-sharings, where for random gates, only the first component of the double-sharing, i.e. the t-sharing is used.

In the computation phase, the actual circuit is computed. For every input gate the secret is shared with Share. Due to the linearity of the used secret-sharing, the linear gates can be computed locally – without communication. Random gates are evaluated simply by picking an unused pre-generated sharing of a random value r. Multiplication gates are evaluated with the help of the pre-generated random double-sharings. Output gates involve a secret reconstruction.

Protocol PassiveMPC**.**

1. PREPARATION PHASE: Invoke DoubleShareRandomPassive$(t, 2t)$ $\lceil \frac{c_M + c_R}{n-t} \rceil$ times in parallel.
2. COMPUTATION PHASE:

   To every random and multiplication gate one of the pre-generated random double-sharings is associated and the circuit is evaluated as follows:

   - INPUT GATE (USER U INPUTS s): Invoke Share(U, s, t) to let the user U share his input s among the players in $\mathcal{P}$.
   - ADDITION/LINEAR GATE: Every $P_i \in \mathcal{P}$ applies the linear function to his respective shares.
   - RANDOM GATE: Pick the t-sharing $[r]_t$ associated with the gate.
   - MULTIPLICATION GATE: Denote the factor sharings as $[x]_t, [y]_t$ and the associated double-sharing as $[r]_{t,2t}$. The product sharing $[z]_t$ is computed by invoking the sub-protocol MultPassive$([x]_t, [y]_t, [r]_{t,2t})$.
   - OUTPUT GATE (OUTPUT $[s]$ TO USER U): Invoke the protocol ReconsPrivPassive$(U, t, [s]_t)$.

Theorem 1 *The MPC protocol* PassiveMPC *evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, communicating $(c_I + 4c_M + 2c_R + c_O)n\kappa + n^2\kappa = \mathcal{O}\big((c_I + c_R + c_M + c_O)n\kappa + n^2\kappa\big)$ bits. The protocol is perfectly secure against a passive adversary corrupting $t < n/2$ players.*

Proof: The security of PassiveMPC follows directly from the security of Share, DoubleShareRandomPassive, MultPassive and ReconsPrivPassive.

The communication complexity of the preparation phase is $\lceil \frac{c_M + c_R}{n-t} \rceil$ times the complexity of DoubleShareRandomPassive, i.e.

$$\begin{aligned}
\lceil \frac{c_M + c_R}{n-t} \rceil 2n^2\kappa &< \lceil \frac{3(c_M + c_R)}{2n} \rceil 2n^2\kappa \\
&< (\frac{3(c_M + c_R)}{2n} + 1)2n^2\kappa \\
&\leq 3(c_M + c_R)n\kappa + 2n^2\kappa
\end{aligned}$$

The communication complexity of the computation phase is

$$c_I\mathsf{Share} + c_M\mathsf{MultPassive} + c_O\mathsf{ReconsPrivPassive} = c_I n\kappa + c_M 2n\kappa + c_O n\kappa.$$

Thus the protocol PassiveMPC communicates less than

$$3(c_M+c_R)n\kappa+2n^2\kappa+c_I n\kappa+c_M 2n\kappa+c_O n\kappa = (c_I+5c_M+3c_R+c_O)n\kappa+2n^2\kappa$$

■

Chapter 4

Active MPC (Without Setup)

4.1 Introduction

In the active model the adversary can make the corrupted parties deviate from the protocol in any desired manner.

Thus, in order to make a passively secure protocol detectable, correctness checks have to be introduced to detect faults caused by actively corrupted parties.[6] For robust protocols, additionally some fault recovery procedures have to be invoked in case of faults.

For example, when invoking the passively secure sharing protocol $\mathsf{Share}(d)$ in the presence of an active adversary, a corrupted dealer P_D might distribute inconsistent shares (lying on a polynomial of a higher degree than d), without the honest players noticing.

The first active MPC protocol perfectly secure for $t < n/3$ (from [BGW88]) solves this problem by presenting a verifiable secret-sharing scheme using two-dimensional polynomials[7], which produces correct sharings even for corrupted dealers. However, this protocol is very inefficient (it requires many invocations to the BA primitives)

[6]Remember that according to Definition 1 a detectable protocol is a passively secure protocol that can produce incorrect output if some players misbehave, however this will be noticed by at least one honest player.

[7]explained in Chapter 7

A more efficient approach (from [HM01]) is to use the passively secure protocol $\mathsf{Share}(d)$, and then to check the correctness of many sharings in parallel by checking the correctness of a blinded random linear combination of these sharings. However, this approach yields a detectable secret-sharing protocol only.

In order to transform a detectable protocol into a robust protocol a general technique – Player Elimination – is presented in [HMP00]. The goal of player elimination is to limit the number of faults the adversary can cause, by localizing and eliminating a pair of players containing at least one corrupted player, every time a fault is detected. Then the computation is repeated with the reduced player set.

Based on the efficient passively-secure protocol presented in the previous chapter and the above ideas from [HM01] and [HMP00], an actively secure protocol for $t < n/3$ with linear communication complexity is constructed in [DN07]. However, this protocol is only statistically secure – there is a negligible error probability due to the probabilistic correctness checks.

In this chapter we present a novel technique which, at the same time, allows to perfectly and very efficiently verify a bunch of sharings and (if the check says that they are correct) to extract a set of (new) correct random sharings given that a sub-set of the original sharings is random. The novel technique is based on so-called *hyper-invertible matrices*, i.e., matrices whose every square sub-matrix is invertible. Applying n sharings to such a matrix results in n sharings with the property that (i) if *any* (up to t) of the inputs sharings are incorrect, then this can be seen in *every* subset of t output sharings, and (ii) if *any* $n-t$ input sharings are uniform random, then *every* subset of size $n-t$ of output sharings is uniform random.

Using hyper-invertible matrices and some techniques from [Bea91a, HMP00, DN07], we construct a perfectly secure multi-party protocol with optimal resilience and linear communication complexity (published in [BH08]).

We will first present a non-robust but fair protocol for non-reactive MPC, constructed from the passively secure protocol (from the previous chapter) using hyper-invertible matrices. Then this protocol will be extended using the circuit randomization technique from [Bea91a] to a non-robust but fair protocol for reactive MPC. Lastly, using the player elimination technique from [HMP00], the fair non-robust protocol for reactive MPC will be transformed into a robust protocol for reactive MPC with linear communication complexity.

4.2 Model

We consider a set $\mathcal{U}$ of users and a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, connected by a complete network of secure (private and authentic) synchronous channels.

The function to be computed is specified as an arithmetic circuit over a finite field $\mathbb{F}$ (with $|\mathbb{F}| > 2n$), with input, addition, multiplication, random, and output gates. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O, respectively.

The adversary can corrupt up to $t < n/3$ players, any number of users and is computationally unbounded, active, adaptive and rushing. The security of our protocols is perfect, i.e., information-theoretic without any error probability.

To every player $P_i \in \mathcal{P}$ a unique, non-zero element $\alpha_i \in \mathbb{F} \setminus \{0\}$ is assigned.

For the ease of presentation, we always assume that the messages sent through the channels are from the right domain — if a player receives a message which is not in the right domain (e.g., no message at all), he replaces it with an arbitrary message from the specified domain.

4.3 Secret Sharing

4.3.1 Definitions and Notation

We define a correct sharing according to Definition 4, i.e. we say that a value s is (correctly) *d-shared* (among the players in $\mathcal{P}$) if every honest player $P_i \in \mathcal{P}$ is holding a share s_i of s, such that there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$ and $p(\alpha_i) = s_i$ for every $P_i \in \mathcal{P}$.[8] The vector $(s_1, \ldots, s_n)$ of shares is called a d-sharing of s, and is denoted by $[s]_d$. A (possibly incomplete) set of shares is called *d-consistent* if these shares lie on a degree d polynomial.

Note that in contrast to the passive case, a correct d-sharing with $n - t \leq d < n$ does not necessarily well-define any value (depending on the actual number of corrupted players).

[8]Where α_i denotes the unique fixed value assigned to P_i.

As in Chapter 3, by saying that the players in $\mathcal{P}$ compute (locally)

$$([y^{(1)}]_{d'}, \ldots, [y^{(m')}]_{d'}) = f([x^{(1)}]_d, \ldots, [x^{(m)}]_d)$$

(for any function $f : \mathbb{F}^m \to \mathbb{F}^{m'}$) we mean that every player P_i applies this function to his shares, i.e. computes

$$(y_i^{(1)}, \ldots, y_i^{(m')}) = f(x_i^{(1)}, \ldots, x_i^{(m)}).$$

Remember that by applying any linear or affine function to correct d-sharings we get a correct d-sharing of the output. However, by multiplying two correct d-sharings we get a correct $2d$-sharing of the product, i.e. $[a]_d[b]_d = [ab]_{2d}$.

In the multiplication protocol, we again use double-sharings as defined in Definition 5: We say that a value x is (d, d')-*shared* among the players $\mathcal{P}$, denoted as $[x]_{d,d'}$, if x is both d-shared and d'-shared.

4.3.2 The Share Protocol

We use two sharing protocols: the protocol Share (from Section 3.3.2) and the protocol VShare. The protocol Share outputs correct sharings if the dealer follows the protocol, whereas the output of VShare is always a correct sharing.

Remember, that the protocol Share allows an honest dealer P_D to correctly d-share a secret s among the players in $\mathcal{P}$, while communicating $n\kappa$ bits. However, this protocol does not ensure that the resulting sharing is correct; a corrupted dealer might distribute totally inconsistent shares. The correctness of sharings must be verified separately.

In order to have a dealer P_D *verifiably share* a value s with degree d the following protocol VShare is invoked. VShare uses a random d-sharing $[r]_d$ which is privately reconstructed towards the dealer (using the protocol ReconsPriv described in the next subsection). The dealer then broadcasts the difference $\delta = s - r$ and the players compute the d-sharing $[s]_d$ of s as $[s]_d = [r]_d + \delta$.

Protocol VShare$(P_D \in (\mathcal{P} \cup \mathcal{U}), s, d, [r]_d)$**.**

1. RECONSTRUCT r: Invoke ReconsPriv$(P_D, d, [r]_d)$ to reconstruct $[r]_d$ towards the dealer P_D.

2. BROADCAST DIFFERENCE: P_D computes and broadcasts (in $\mathcal{P}$) the difference $\delta = s - r$.
3. COMPUTE AND OUTPUT: The players in $\mathcal{P}$ compute and output the d-sharing $[s]_d = [r]_d + \delta$.

The above protocol is robust for $d < n - 2t$ (as then ReconsPriv is robust). The outputted sharing is a correct d-sharing of $s' = r + \delta$ if $[r]_d$ is a correct d-sharing. For an honest dealer P_D additionally holds $s' = r + \delta = r + s - r = s$. VShare is private if r is a secret random value (as then $\delta = s - r$ does not leak any information to the adversary). The communication complexity of VShare is $n\kappa + \mathcal{BC}(\kappa)$.[9]

4.3.3 The Reconstruct Protocol

We use two reconstruction protocols: one for private and one for public reconstruction. Both can be either robust or only detectable – depending on the degree of the sharings to be reconstructed.

In the private reconstruction protocol the players simply send their shares to the receiver P_R (a player or a user), who interpolates the secret (if possible).

Protocol ReconsPriv$(P_R \in (\mathcal{P} \cup \mathcal{U}), d, [s]_d)$.

1. SEND SHARES: Every player $P_i \in \mathcal{P}$ sends his share s_i of s to P_R.
2. INTERPOLATE AND OUTPUT: If there exists a degree-d polynomial $p(\cdot)$ such that at least $d+1+t$ of the received shares lie on it, i.e. $p(\alpha_i) = s_i$, then P_R computes and outputs the secret $s = p(0)$. Otherwise P_R gets unhappy (sets his happy-bit to "unhappy").[10]

Lemma 1 *For $d < n - 2t$, the protocol* ReconsPriv robustly *reconstructs $[s]_d$ towards P_R. For $d < n - t$,* ReconsPriv detectably *reconstructs $[s]_d$ towards P_R (i.e., P_R either outputs s or gets unhappy, where the latter only happens when some players deviate from the protocol).* ReconsPriv *communicates $n\kappa$ bits.*

[9]Where $\mathcal{BC}(\cdot)$ denotes the number of broadcasted bits. Remember that in this model broadcasting κ bits corresponds to communicating $n^2\kappa$ bits.

[10]Remember that every player has an internal state (the happy-bit) signalizing whether or not he detected a fault in the computation.

The public reconstruction protocol ReconsPubl takes $T = n - 2t = \Omega(n)$ correct d-sharings $[s_1]_d, \ldots, [s_T]_d$ and publicly (to all players in $\mathcal{P}$) outputs the (correct) values $s_1, \ldots, s_T$, or fails (with at least one honest player being unhappy). In ReconsPubl we use the idea of [DN07]: first the T sharings $[s_1]_d, \ldots, [s_T]_d$ are expanded (using a linear error-correcting code) to n sharings $[u_1]_d, \ldots, [u_n]_d$,[11] each of which is reconstructed towards *one* player in $\mathcal{P}$ (using ReconsPriv). Then, every $P_i \in \mathcal{P}$ sends his reconstructed value u_i to every other player in $\mathcal{P}$, who tries to decode (with error correction) the received code word $(u_1, \ldots, u_n)$ to $s_1, \ldots, s_T$. ReconsPubl communicates $\mathcal{O}(n^2\kappa)$ bits to reconstruct $T = \Omega(n)$ sharings.

Protocol ReconsPubl$(d, [s_1]_d, \ldots, [s_T]_d)$.

1. EXPAND: For $j = 1, \ldots, n$ the players in $\mathcal{P}$ (locally) compute:

$$[u_j]_d = [s_1]_d + [s_2]_d\beta_j + [s_3]_d\beta_j^2 + \ldots + [s_T]_d\beta_j^{T-1}$$

   for some fixed distinct values $\beta_1, \ldots, \beta_n \in \mathbb{F} \setminus \{0\}]$.
2. RECONSTRUCT: For every $P_i \in \mathcal{P}$, ReconsPriv$(P_i, d, [u_i]_d)$ is invoked to reconstruct $[u_i]_d$ towards P_i.
3. SEND: Every $P_i \in \mathcal{P}$ sends u_i (or $\perp$ if unhappy) to every $P_j \in \mathcal{P}$.
4. COMPUTE AND OUTPUT: $\forall P_i \in \mathcal{P}$: If P_i received at least $T+t$ $(T-1)$-consistent values (in the previous step), he computes (using Lagrange Interpolation) $s_1, \ldots, s_T$. Otherwise he gets unhappy.

Lemma 2 *For $d < n - 2t$, the protocol* ReconsPubl *robustly reconstructs $[s_1]_d, \ldots, [s_T]_d$ towards all players in $\mathcal{P}$. For $d < n-t$,* ReconsPubl *detectably reconstructs $[s_1]_d, \ldots, [s_T]_d$ towards all players in $\mathcal{P}$ (i.e., every $P_i \in \mathcal{P}$ either outputs $s_1, \ldots, s_T$ or gets unhappy, where the latter only happens when some players are faulty).*

ReconsPubl *communicates $2n^2\kappa$ bits to reconstruct $T = n-2t > n/3$ sharings. Thus the amortized communication complexity per reconstructed sharing is less than $6n\kappa$ bits.*

[11] for this we interpret $s_1, \ldots, s_T$ as coefficients of a degree $T-1$ polynomial and $u_1, \ldots, u_n$ as evaluations of this polynomial at n fixed positions $\beta_1, \ldots, \beta_n$.

4.4 Computing of Affine Functions

The computation of affine functions is local. As in the passive case (Section 3.4), the players apply the function to be computed directly to their shares.

4.5 Generating Random Sharings

The non-robust actively-secure generation of random sharings starts similarly as in the passively-secure protocol from Section 3.5.2, by having every player d-share his random contribution with the protocol Share. However in the presence of the active adversary, up to t of the contributed sharings can be incorrect. Hence the correctness of the sharings has to be verified.

In the past years, this was done by checking a blinded random linear combination of the sharings. If the check succeeded (and thus the original sharings were correct with overwhelming probability), the $n-t$ secret random sharing were extracted using super-invertible matrices. While this approach is very efficient, it has two drawbacks: it requires a secure generation of a random challenge (which typically again requires a generation of random sharings) and it has an error-probability.

In this section we present a new technique which at the same time allows to perfectly (and with similar efficiency as before) check the correctness of the contributed sharings and (if the check says that they are o.k.) to extract $\mathcal{O}(n)$ correct secret random sharings.

In the following we introduce hyper-invertible matrices (the basis of our novel technique), and then show how to use them in order to efficiently detectably generate random sharings with perfect security.

4.5.1 Hyper-Invertible Matrices (HIM)

4.5.1.1 Definition

A hyper-invertible matrix (HIM) is a matrix of which every (non-trivial) square sub-matrix is invertible.

Definition 6 *An r-by-c matrix M is* hyper-invertible *if for any index sets $R \subseteq \{1, \ldots, r\}$ and $C \subseteq \{1, \ldots, c\}$ with $|R| = |C| > 0$, the matrix M_R^C is invertible, where M_R denotes the matrix consisting of the rows $i \in R$ of M, M^C denotes the matrix consisting of the columns $j \in C$ of M, and $M_R^C = \big(M_R\big)^C$.*

4.5.1.2 Construction

We present a construction of a hyper-invertible n-by-n matrix M over a finite field $\mathbb{F}$ with $|\mathbb{F}| \geq 2n$. A hyper-invertible r-by-c matrix can be extracted as a sub-matrix of such a matrix with $n = \max(r, c)$.

Construction 1 *Let $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ denote fixed distinct elements in $\mathbb{F}$, and consider the function $f : \mathbb{F}^n \to \mathbb{F}^n$, mapping $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$ such that the points $(\beta_1, y_1), \ldots, (\beta_n, y_n)$ lie on the polynomial $g(\cdot)$ of degree $n-1$ defined by the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n)$. Due to the linearity of Lagrange interpolation, f is linear and can be expressed as a matrix $M = \{\lambda_{i,j}\}_{i=1,\ldots,n}^{j=1,\ldots n}$, where $\lambda_{i,j} = \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{\beta_i - \alpha_k}{\alpha_j - \alpha_k}$.*

Lemma 3 *Construction 1 yields a hyper-invertible n-by-n matrix M.*

Proof: We have to show that for any index sets $R, C \subseteq \{1, \ldots, n\}$ with $|R| = |C| > 0$, M_R^C is invertible. As $|R| = |C|$, it is sufficient to show that the mapping defined by M_R^C is surjective, i.e., for every $\vec{y}_R$ there exists an $\vec{x}_C$ such that $\vec{y}_R = M_R^C \vec{x}_C$. Equivalently, we show that for every $\vec{y}_R$ there exists an $\vec{x}$ such that $\vec{y}_R = M_R \vec{x}$ and $\vec{x}_{\overline{C}} = \vec{0}$, where $\overline{C} = \{1, \ldots, n\} \setminus C$. Remember that M is defined such that the points $(\alpha_1, x_1), \ldots, (\alpha_n, x_n), (\beta_1, y_1), \ldots, (\beta_n, y_n)$ lie on a polynomial $g(\cdot)$ of degree $n-1$. Given the n points $\big\{(\alpha_j, 0)\big\}_{j \notin C}$ and $\big\{(\beta_i, y_i)\big\}_{i \in R}$, the polynomial $g(\cdot)$ can be determined by Lagrange interpolation, and $\vec{x}_C$ can be computed linearly from $\vec{y}_R$. Hence, M_R^C is invertible. ■

4.5.1.3 Properties

The mappings defined by hyper-invertible matrices have a very nice symmetry property: Any subset of n input/output values can be expressed as a linear function of the remaining n input/output values:

Lemma 4 *Let M be a hyper-invertible n-by-n matrix and $(y_1, \ldots, y_n) = M(x_1, \ldots, x_n)$. Then for any index sets $A, B \subseteq \{1, \ldots, n\}$ with $|A| + |B| = n$, there exists an invertible linear function $f : \mathbb{F}^n \to \mathbb{F}^n$, mapping the values $\{x_i\}_{i \in A}, \{y_i\}_{i \in B}$ onto the values $\{x_i\}_{i \notin A}, \{y_i\}_{i \notin B}$.*

Proof: We have $\vec{y} = M\vec{x}$ and $\vec{y}_B = M_B\vec{x} = M_B^A\vec{x}_A + M_B^{\overline{A}}\vec{x}_{\overline{A}}$. Due to hyper-invertibility, $M_B^{\overline{A}}$ is invertible, and $\vec{x}_{\overline{A}} = \big(M_B^{\overline{A}}\big)^{-1}\big(\vec{y}_B - M_B^A\vec{x}_A\big)$. $\vec{y}_{\overline{B}}$ can be computed similarly. ■

4.5.2 Generating Random Sharings Using HIM

The following non-robust protocol ShareRandom(d) either generates $T = n - 2t$ independent secret random values $r_1, \ldots, r_T$, each independently d-shared in $\mathcal{P}$, or fails with at least one honest player being unhappy.

The generation of the random d-sharings employs hyper-invertible matrices: First, every player $P_i \in \mathcal{P}$ selects and d-shares a random value s_i. Then, the players compute d-sharings of the values r_i, defined as $(r_1, \ldots, r_n) = M(s_1, \ldots, s_n)$, where M is a hyper-invertible n-by-n matrix. $2t$ of the resulting d-sharings are reconstructed, each towards a different player, who verifies the correctness of the d-sharing (and gets unhappy in case of a fault). The remaining $n - 2t = T$ sharings are outputted. This procedure guarantees that if all honest players are happy, then at least n sharings are correct (the $n - t$ sharings inputted by honest players, as well as the t sharings verified by honest players), and due to the hyper-invertibility of M, *all* $2n$ sharings must be correct (the remaining sharings can be computed linearly from the good sharings). Furthermore, the outputted sharings are random and unknown to the adversary.

Protocol ShareRandom(d)**.**

1. SECRET SHARE: Every player $P_i \in \mathcal{P}$ chooses a random $s_i \in_R \mathbb{F}$ and acts as a dealer in Share (P_i, s_i, d) to distribute the shares among the players in $\mathcal{P}$ – resulting in a sharing $[s_i]_d$.
2. APPLY M: The players in $\mathcal{P}$ (locally) compute

$$\big([r_1]_d, \ldots, [r_n]_d\big) = M\big([s_1]_d, \ldots, [s_n]_d\big).$$

   In order to do so, every player computes his share of each r_j as linear combination of his shares of the s_i-values.

3. CHECK: For $i = T+1, \ldots, n$, every $P_j \in \mathcal{P}$ sends his share of $[s_i]_d$ to P_i, who checks that *all* n shares are d-consistent, i.e. P_i checks that all shares indeed lie on a degree-d polynomial. If not P_i gets unhappy.
4. OUTPUT: The remaining T sharings $[r_1]_d, \ldots, [r_T]_d$ are outputted.

Lemma 5 *If all players follow the protocol, then the protocol succeeds (i.e., all honest players remain happy). If* $\mathsf{ShareRandom}(d)$ *succeeds, it outputs* $T = n - 2t$ *correct and random* d*-sharings, unknown to the adversary.*

$\mathsf{ShareRandom}$ *communicates* $\frac{5}{3}n^2\kappa$ *bits to generate* $n-2t$ *random sharings. The amortized communication complexity per sharing is less than* $5n\kappa$ *bits.*

Proof: CORRECTNESS: Assume that all honest players remain happy during the protocol. Then for all honest P_i with $i \in \{T+1, \ldots, n\}$, the sharing of r_i checked by P_i in Step 3 is a correct d-sharing. As $T = n - 2t$, there are at least t correct sharings of the values r_k. Furthermore, every sharing of an s_i distributed by an honest P_i in Step 1 is a correct d-sharing. Thus there are at least $n-t$ correct sharings of the values s_k. Given these (at least) n correct d-sharings, the sharings of *all* other values s_k and r_k can be computed linearly. As a linear combination of correct d-sharings is again a correct d-sharing, it follows that all values $s_1, \ldots, s_n, r_1, \ldots, r_n$ are correctly d-shared.

PRIVACY: The adversary knows (at most) t of the input sharings s_k (those provided by corrupted players), and t of the output sharings r_k (with $k > T$, those reconstructed towards corrupted players). When fixing these $2t$ sharings, then there exists a bijective mapping between any other honest $T = n - 2t$ input sharings (independent of the $2t$ sharings known to the adversary) and the first T output sharings (Lemma 4), hence the sharings $[r_1]_d, \ldots, [r_T]_d$ are uniformly at random, unknown to the adversary.

COMMUNICATION: The protocol communicates

$$n\mathsf{Share} + 2tn\kappa < n^2\kappa + 2/3n^2\kappa = 5/3n^2\kappa$$

bits to generate $T = n - 2t > n/3$ random sharings. Thus the amortized communication complexity per generated sharing is less than $5n\kappa$ bits. ■

4.6 Multiplication

The detectable multiplication protocol Mult is based on the passive multiplication protocol MultPassive (Section 3.6.2): it multiplies two d-shared values using one $(d, 2d)$-sharing of a secret random value. We first describe the detectable protocol for generating random double-sharings DoubleShareRandom and then the detectable protocol for multiplying values given random double-sharings.

The following protocol DoubleShareRandom works along the lines of ShareRandom (from Section 4.5.2) using a hyper-invertible n-by-n matrix M. It generates $T = n-2t$ correct (d, d')-sharings of secret random values (or fails with at least one honest player being unhappy).

Protocol DoubleShareRandom(d, d')**.**

1. SECRET SHARE: Every player $P_i \in \mathcal{P}$ chooses a random $s_i \in_R \mathbb{F}$ and acts as a dealer in Share (P_i, s_i, d) and in Share (P_i, s_i, d') to distribute the shares of $[s_i]_d$ and $[s_i]_{d'}$ among the players in $\mathcal{P}$ – resulting in a double-sharing $[s_i]_{d,d'}$.
2. APPLY M: The players in $\mathcal{P}$ (locally) compute
$$\big([r_1]_{d,d'}, \ldots, [r_n]_{d,d'}\big) = M\big([s_1]_{d,d'}, \ldots, [s_n]_{d,d'}\big).$$
In order to do so, every player computes his double-share of each r_j as linear combination of his double-shares of the s_i-values.
3. CHECK: For $i = T + 1, \ldots, n$, every $P_j \in \mathcal{P}$ sends his double-share $[s_i]_{d,d'}$ to P_i, who checks that *all* n double-shares define a correct double-sharing of some value s_i. More precisely, P_i checks that all d-shares indeed lie on a polynomial $g(\cdot)$ of degree d, and that all d'-shares indeed lie on a polynomial $g'(\cdot)$ of degree d', and that $g(0) = g'(0)$. If any of the checks fails, P_i gets unhappy.
4. OUTPUT: The remaining T double-sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are outputted.

Lemma 6 *If all players follow the protocol, then the protocol succeeds (i.e., all honest players remain happy). If* DoubleShareRandom(d, d') *succeeds, it outputs $T = n-2t$ correct and random (d, d')-sharings, unknown to the adversary.*

DoubleShareRandom *communicates $\frac{10}{3}n^2\kappa$ bits to generate $n - 2t$ double-sharings. The amortized complexity per double-sharing is less than $10n\kappa$ bits.*

The following multiplication protocol Mult is almost identical to the passive multiplication protocol MultPassive from Section 3.6.2. However, it proceeds $T = n - 2t$ multiplications in parallel (as the used sub-protocol ReconsPubl has linear communication complexity only if reconstructing $T = n - 2t$ values in parallel).

Protocol Mult($[x_1]_d, [y_1]_d, [r_1]_{d,2d} \dots, [x_T]_d, [y_T]_d, [r_T]_{d,2d}$) **.**

1. MULTIPLY SHARES: For $k = 1, \dots T$ the players compute (locally) the $2d$-sharing $[z_k]_{2d}$ of $z_k = x_k y_k$ as $[z_k]_{2d} = [x_k]_d [y_k]_d$ (by every player computing the product of his shares).
2. REDUCE DEGREE
   - 2.1 For $k = 1, \dots T$ the players compute (locally) the $2d$-sharing of the difference $\delta_k = z_k - r_k$ as $[\delta_k]_{2d} = [z_k]_{2d} - [r_k]_{2d}$.
   - 2.2 Invoke ReconsPubl($[\delta_1]_{2d}, \dots, [\delta_T]_{2d}$ to reconstruct the differences $\delta_1, \dots, \delta_T$ towards every player in $\mathcal{P}$.
   - 2.3 For $k = 1, \dots T$ the players compute (locally) the d-sharing $[z_k]_d = [r_k]_d + \delta_k$.
3. OUTPUT: The players output the d-sharings $[z_1]_d, \dots, [z_T]_d$.

The protocol Mult is robust for $2d < n - 2t$ and detectable for $2d < n - t$. The security of Mult follows directly from the security of MultPassive and ReconsPubl. The protocol Mult communicates at $2n^2\kappa$ bits to multiply $n - 2t > n/3$ pairs, thus the amortized communication complexity per one multiplication is less than $6n\kappa$ bits.

4.7 Fault Detection

The following fault detection protocol allows the players to reach agreement on whether or not all players are happy.

Protocol FaultDetection**.**

1. DISTRIBUTE HAPPY BITS: Every $P_i \in \mathcal{P}$ sends his happy-bit to every $P_j \in \mathcal{P}$, who gets unhappy if at least one P_i claims to be unhappy.
2. FIND AGREEMENT: The players in $\mathcal{P}$ run a consensus protocol with every player inputting his happy-bit.

3. OUTPUT: If the consensus of the previous step outputs "happy", output "succeeded" otherwise output "failed".

Note that the honest players always agree on the output of FaultDetection and if at least one honest player is unhappy at the beginning of the protocol, then the output is "failed "(regardless of the behavior of the corrupted players). If all honest players start the protocol being happy and all players follow the protocol then FaultDetection outputs "succeeded". However the adversary can always cause the output to be "failed", even if all honest players are happy at the beginning of the protocol.

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

The above protocol will be invoked at the end of the non-robust parts of the computation to determine whether the computation was correct and its outputs can be used in the following computation.

4.8 The Fair SFE Protocol

Using the sub-protocols presented by now, we are able to construct a non-robust but fair protocol for non-reactive MPC (but not yet for fair reactive MPC [12]).

4.8.1 Overview

The protocol proceeds in three phases: the preparation phase, the computation phase (without output gates) and the output phase.

In the preparation phase, secret random $t, 2t$-double-sharings are generated (in parallel), one for every multiplication gate. Furthermore, for every random gate as well as for every input gate, a t-sharing of a random r is generated. For the sake of simplicity, we generate $c_M + c_R + c_I$ random double-sharings, where for random and input gates, only the first component (the t-sharing) is used.

In the computation phase, the actual circuit is computed. Input gates are robustly evaluated with the help of a pre-shared random value r. Due

[12]Remember that for $d = t$ and $t < n/3$ the multiplication protocol is non-robust. Hence if there is any multiplication gate strict after an output gate, the adversary can cause the computation to fail after receiving output.

to the linearity of the used secret-sharing, the linear gates can be computed locally – without communication. Random gates are evaluated simply by picking an unused pre-shared random value r.

Multiplication gates are evaluated non-robustly with help of one pre-generated double-sharing (at the cost of one non-robust public reconstruction of a $2t$-sharing). For the sake of efficiency, we evaluate $T = n - 2t$ multiplication gates of the same multiplicative depth at once (such that we can publicly reconstruct T sharings at once).[13]

In the output phase the output gates are evaluated using secret reconstruction which is robust for $d = t$.

The preparation and the computation phase can both fail in case of faults, however the output phase is robust, given that preparation and computation succeeded (all players remained happy).

4.8.2 Preparation Phase

The protocol PreparationPhase first invokes the non-robust protocol DoubleShareRandom$(t, 2t)$ (many times in parallel) to let the players generate a bunch of secret random double-sharings. Then FaultDetection is invoked to let the players agree on whether or not the generation of the double-sharings succeeded.

Protocol PreparationPhase**.**

1. GENERATION: Invoke DoubleShareRandom$(t, 2t)$ $\lceil \frac{c_I + c_M + c_R}{n-2t} \rceil$ times in parallel.
2. FAULT DETECTION: Invoke FaultDetection.
3. OUTPUT: If the output of the previous step is "succeeded", the double-sharings generated in Step 1 are outputted. Otherwise, the preparation phase failed.

The players always agree on whether PreparationPhase succeeded or failed. If all players follow the protocol, then PreparationPhase succeeds. If PreparationPhase succeeds, then it outputs $c_I + c_M + c_R$ correct $(t, 2t)$-sharings of independent secret random values. The communication complexity of PreparationPhase is less than $10(c_I + c_M + c_R)n\kappa + 5n^2\kappa + \mathcal{BA}(1)$, which amounts to $\mathcal{O}((c_I n + c_M n + c_R n + n^2)\kappa)$ bits of communication.

[13]The multiplicative depth of a gate is the maximum number of multiplication gates on any path from input/random gates to this gate.

4.8.3 Computation Phase

In the computation phase, the actual circuit (except for the output gates) is computed.

At the end of the computation phase FaultDetection is invoked to let the players agree on whether or not all players are happy.

Protocol ComputationPhase**.**

1. CIRCUIT EVALUATION: To every input, random and multiplication gate, one of the pre-generated random double-sharings is associated and the gates of the circuit are evaluated as follows:
   - INPUT GATE (USER U INPUTS s): Let $[r]_t$ denote the associated random sharing. The protocol VShare$(U, t, s, [r]_t)$ is invoked to let the user U verifiably t-share his input s.
   - ADDITION/LINEAR GATE: Every $P_i \in \mathcal{P}$ applies the linear function on his respective shares.
   - RANDOM GATE: Pick the sharing $[r]_t$ associated with the gate.
   - MULTIPLICATION GATE: Up to $T = n - 2t$ multiplication gates are processed simultaneously. Denote the factor sharings as $\big([x_1]_t, [y_1]_t\big), \ldots, \big([x_T]_t, [y_T]_t\big)$, and the associated double-sharings as $[r_1]_{t,2t}, \ldots, [r_T]_{t,2t}$. The product sharings $[z_1]_t, \ldots, [z_T]_t$ are computed by invoking the sub-protocol Mult.
2. FAULT DETECTION: Invoke FaultDetection.
3. OUTPUT: If FaultDetection outputs "succeeded", output the computed t-sharing of the circuit output(s).

The players always agree on whether ComputationPhase succeeded or failed. The following holds given that the pre-generated double-sharings are correct, random and unknown to the adversary: If all players follow the protocol, then ComputationPhase succeeds (completeness). If ComputationPhase succeeds, then it outputs correct t-sharing of the correct circuit output(s) (correctness). ComputationPhase leaks no information to the adversary (privacy).

The communication complexity of ComputationPhase is less than $c_I(n\kappa + \mathcal{BA}(\kappa)) + 6c_M n\kappa + 2D_M n^2\kappa + n\kappa + \mathcal{BA}(1)$, which amounts to $\mathcal{O}\big((c_I n^2 + c_M n + D_M n^2 + n^2)\kappa\big)$ bits of communication, where D_M denotes the multiplicative depth of the circuit.

4.8.4 Output Phase

In the output phase, every output gate is evaluated robustly by reconstructing the output sharing towards the designated recipient with the protocol ReconsPriv.

Protocol OutputPhase**.**

- OUTPUT GATE (OUTPUT $[s]$ TO USER U): Invoke ReconsPriv$(U, t, [s]_t)$.

For every output gate holds: If the sharing $[s]$ is a correct t-sharings of some value s, then the recipient outputs s. The communication complexity of the output phase is $c_O n\kappa$.

4.8.5 Main Protocol

The main protocol starts by invoking the detectable protocol PreparationPhase which either generates a bunch of correctly shared secret random double-sharings or fails (and the players agree on what is the case). If PreparationPhase succeeds, the circuit (except for the output gates) is detectably evaluated with ComputationPhase using the pre-generated random double-sharings. At the end of ComputationPhase the players again agree whether or not the protocol succeeded, and if yes, the outputs are robustly reconstructed in OutputPhase.

Protocol MainSFE**.**

0. Every $P_i \in \mathcal{P}$ sets his happy-bit to "happy".
1. PREPARATION PHASE: Invoke PreparationPhase.
2. COMPUTATION PHASE: If PreparationPhase succeeded invoke ComputationPhase.
3. OUTPUT PHASE: If OutputPhase succeeded invoke OutputPhase.

Theorem 2 *The protocol* MainSFE *non-robustly but fairly evaluates a function with c_I input, c_R random, c_M multiplication, and c_O output gates, communicating $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2)\kappa + c_I\, \mathcal{BA}(\kappa) + n^2 + \mathcal{BA}(1)\big)$ bits, which amounts to $\mathcal{O}\big((c_I n^2 + c_R n + c_M n + c_O n + D_M n^2 + n^2)\kappa\big)$ bits, where D_M denotes the multiplicative depth of the circuit. The protocol is perfectly secure against an adaptive active adversary corrupting $t < n/3$ players.*

The fairness of this protocol follows from the fact that the output phase (the only part of the computation where the adversary gets some information) is robust, given that the preparation and the computation phase succeeded (which the players have to agree on, before going on to the output phase). To obtain a fair protocol for on-going computations (where an output gate might be followed by an multiplication gate), we have to make the computation of multiplication gates robust as well. To achieve this, we use the Circuit randomization technique [Bea91a]. This technique allows us to transfer "all the non-robust stuff" into the preparation phase (where we only work with random values completely independent of the inputs) such that the actual evaluation of the circuit is fully robust (consisting of reconstructions of t-sharings only).

4.9 Circuit Randomization

Circuit randomization [Bea91a] allows to compute a sharing $[z]_d$ of the product z of two factors x and y, shared as $[x]_d$ and $[y]_d$, at the costs of two public reconstructions, when a pre-shared random triple $\big([a]_d, [b]_d, [c]_d\big)$ with $c = ab$ is available. This technique allows to first prepare c_M shared multiplication triples $\big([a]_d, [b]_d, [c]_d\big)$, and then to evaluate a circuit with c_M multiplications by a sequence of public reconstructions.

The trick of circuit randomization is that $z = xy$ can be expressed as

$$z = \big((x-a)+a\big)\big((y-b)+b\big),$$

hence

$$z = \delta_x\delta_y + \delta_x b + a\delta_y + c,$$

where (a, b, c) is a multiplication triple and $\delta_x = x - a$ and $\delta_y = y - b$. For a random multiplication triple, δ_x and δ_y are random values independent of x and y, hence a sharing $[z]$ can be linearly computed as

$$[z]_d = [\delta_x\delta_y]_d + \delta_x[b]_d + \delta_y[a]_d + [c]_d,$$

by reconstructing $[\delta_x]_d = [x]_d - [a]_d$ and $[\delta_y]_d = [y]_d - [b]_d$.

4.10 The Fair MPC Protocol

With help of the circuit randomization technique, we are now able to convert the previous protocol for non-robust but fair non-reactive MPC to a non-robust but fair protocol for reactive MPC.

4.10.1 Overview

The protocol proceeds in two phases: the non-robust preparation phase and the computation phase.

In the preparation phase, t-sharings of secret random multiplication triples are non-robustly generated, one for every multiplication gate. Furthermore, for every random gate as well as for every input gate, a t-sharing of a random r is generated. For the sake of simplicity, we generate $c_M + c_R + c_I$ random multiplication triples, where for random and input gates, only the first component of the triple is used.

In the computation phase, the actual circuit (including the output gates) is robustly evaluated. Input gates are evaluated with the help of a pre-shared random value r. Linear gates are computed locally – without communication. Random gates are evaluated simply by picking an unused pre-shared random value r. Multiplication gates are evaluated with the help of one pre-generated random multiplication triple (at the cost of two public reconstructions of t-sharings). For the sake of efficiency, we evaluate up to $T/2$ multiplication gates at once (if possible), such that we can publicly reconstruct T sharings at once. Output gates are evaluated using a secret reconstruction.

Note that the preparation phase can fail in case of faults, however the computation phase is robust, given that the preparation phase succeeded (which the player agree on before going on to the computation phase).

4.10.2 Preparation Phase

We first present the non-robust protocol GenerateTriples which either generates $T = n-2t$ correctly d-shared multiplication triples unknown to the adversary or fails with at least one honest player being unhappy.

GenerateTriples generates T pairs of random d-sharings $([a_1]_d, [b_1]_d), \dots, ([a_T]_d, [b_T]_d)$ and T random double-sharings. Then the protocol Mult is invoked to compute the products $[c_1]_d = [a_1]_d[b_1]_d, \dots, [c_T]_d = [a_T]_d[b_T]_d$ with the help of the double-sharings.

Protocol GenerateTriples(d).

1. GENERATE RANDOM a AND b: Invoke ShareRandom(d) two times in parallel to generate $T = n - 2t$ random d-sharings $[a_1]_d, \ldots, [a_T]_d$ and T random d-sharings $[b_1]_d, \ldots, [b_T]_d$.
2. MULTIPLY a AND b:
   2.1 GENERATE RANDOM DOUBLE SHARING $[r]_{d,2d}$: Invoke DoubleShareRandom$(d, 2d)$ to generate T random double sharings $[r_1]_{d,2d}, \ldots, [r_T]_{d,2d}$
   2.2 MULTIPLY a AND b USING $[r]_{d,2d}$: Invoke Mult$([a_1]_d, [b_1]_d, [r_1]_{d,2d}, \ldots, [a_T]_d, [b_T]_d, [r_T]_{d,2d})$ to compute the d-sharings of the products $c_1 = a_1 b_1, \ldots, c_T = a_T b_T$.
3. OUTPUT: The players output the d-sharings of the T triples $([a_1]_d, [b_1]_d, [c_1]_d), \ldots, ([a_T]_d, [b_T]_d, [c_T]_d)$.

Lemma 7 *For $2d < n - t$ the protocol* GenerateTriples(d) *has the following properties: If all players follow the protocol, then the protocol succeeds (completeness). If the protocol succeeds, than it outputs $T = n - 2t$ independent correct d-sharings $([a_1]_d, [b_1]_d, [c_1]_d), \ldots, ([a_T]_d, [b_T]_d, [c_T]_d)$ with a_i, b_i being independent random values and $c_i = a_i b_i$ (correctness). The adversary gets no information on the triples $(a_1, b_1, c_1), \ldots, (a_T, b_T, c_T)$ (privacy).*

The communication complexity of GenerateTriples(d) *is at most $\frac{26}{3} n^2 \kappa$. The amortized communication complexity per triple is at most $26 n \kappa$.*

The following protocol PreparationPhase first invokes the non-robust protocol GenerateTriples(t) (many times in parallel) to let the players generate a bunch of secret random multiplication triples. Then FaultDetection is invoked to let the players agree on whether or not the generation of the triples succeeded.

Protocol PreparationPhase**.**

0. Every $P_i \in \mathcal{P}$ sets his happy-bit to "happy".
1. GENERATION: Invoke GenerateTriples(t) $\lceil \frac{c_I + c_M + c_R}{n - 2t} \rceil$ times in parallel.
2. FAULT DETECTION: Invoke FaultDetection.
3. OUTPUT: If the output of the previous step is "succeeded" the multiplication triples generated in Step 1 are outputted.

The players always agree on whether PreparationPhase succeeded or failed. If all players follow the protocol, then PreparationPhase succeeds (completeness). If PreparationPhase succeeds, then it outputs $\ell = c_I + c_M + c_R$ independent correct t-sharings $([a_1]_t, [b_1]_t, [c_1]_t), \ldots, ([a_\ell]_t, [b_\ell]_t, [c_\ell]_t)$ with a_i, b_i being independent random values and $c_i = a_i b_i$ (correctness). The adversary gets no information on the triples $(a_1, b_1, c_1), \ldots, (a_\ell, b_\ell, c_\ell)$ (privacy).

The communication complexity of PreparationPhase is less than $26(c_I + c_M + c_R)n\kappa + 10n^2\kappa + \mathcal{BA}(1)$ which amounts to $\mathcal{O}\big((c_I + c_R + c_M)n\kappa + n^2\kappa\big)$ bits of communication.

4.10.3 Computation Phase

We first present the protocol MultiplicationGate which robustly computes up to $\lfloor T/2 \rfloor$ (with $T = n - 2t$) products in parallel using pre-generated multiplication triples.

Denote the factor sharings as $\big([x_1]_d, [y_1]_d\big), \ldots, \big([x_{T/2}]_d, [y_{T/2}]_d\big)$ and the sharings of the triples as $([a_1]_d, [b_1]_d, [c_1]_d), \ldots, ([a_{T/2}]_d, [b_{T/2}]_d, [c_{T/2}]_d)$. Then the products $[z_1]_d, \ldots, [z_{T/2}]_d$ are computed as follows:

Protocol MultiplicationGate(d)**.**

1. Compute the Differences: For $k = 1, \ldots, T/2$, the players compute $[d_k]_d = [x_k]_d - [a_k]_d$ and $[e_k]_d = [y_k]_d - [b_k]_d$.
2. Reconstruct the Differences: Invoke ReconsPubl to publicly reconstruct the T d-sharings of $(d_1, e_1), \ldots, (d_{T/2}, e_{T/2})$. Note that this is robust, as long as $d < n - 2t$.
3. Compute and Output the Products: For $k = 1, \ldots, T/2$, the players compute the product sharings $[z_k]_d = d_k e_k + d_k [b_k]_d + e_k [a_k]_d + [c_k]_d$.

Lemma 8 *For $d < n - 2t$ the protocol* MultiplicationGate(d) *perfectly securely computes up to $\lfloor T/2 \rfloor$ multiplications, given $\lfloor T/2 \rfloor$ pre-shared correct random multiplication triples. The communication complexity of* MultiplicationGate *is $2n^2\kappa$ bits.*

As the computation phase involves t-sharings only and $t < n - 2t$, the evaluation of the multiplication gates is robust. Hence all circuit gates can be evaluated robustly, i.e once the computation phase starts, the adversary cannot prevent the honest players from receiving their output. Thus we can evaluate output gates at any point in the computation (according to the circuit).

Protocol ComputationPhase**.**

To every input, random and multiplication gate one pre-generated random triple is associated and the gates of the circuit are evaluated as follows:

- INPUT GATE (USER U INPUTS s): Let $[r]_t$ denote the associated random sharing. The protocol VShare$(U, t, s, [r]_t)$ is invoked to let the user U verifiably t-share his input s.
- ADDITION/LINEAR GATE: Every $P_i \in \mathcal{P}$ applies the linear function on his respective shares.
- RANDOM GATE: Pick the sharing $[r]_t$ associated with the gate.
- MULTIPLICATION GATE: Up to $\lfloor T/2 \rfloor$ multiplication gates are processed in parallel by invoking the protocol MultiplicationGate(t) (using the associated multiplication triples $([a_1]_t, [[t]b_1], [c_1]_t), \dots, ([a_{T/2}]_t, [b_{T/2}]_t, [c_{T/2}]_t)$).
- OUTPUT GATE (OUTPUT $[s]$ TO USER U): Invoke ReconsPriv$(U, t, [s]_t)$.

Lemma 9 *The protocol* ComputationPhase *perfectly securely evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, given $c_I + c_R + c_M$ pre-shared correct, random multiplication triples, with communicating $\mathcal{O}\big((c_I n + c_M n + c_O n + D_M n^2)\kappa + c_I\,\mathcal{BA}(\kappa)\big)$ bits, where D_M denotes the multiplicative depth of the circuit.*

4.10.4 Main Protocol

The protocol NonRobusteMPC consists of a non-robust preparation phase and a robust computation phase which is invoked only if the preparation phase succeeded.

Protocol NonRobusteMPC**.**

1. PREPARATION PHASE: Invoke PreparationPhase.
2. COMPUTATION PHASE: If PreparationPhase succeeded invoke ComputationPhase, otherwise abort the computation.

Theorem 3 *The protocol* NonRobusteMPC *non-robustly but fairly evaluates a (reactive) circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, communicating $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2 + n^2)\kappa + c_I\,\mathcal{BA}(\kappa) + \mathcal{BA}(1)\big)$ bits, which amounts to $\mathcal{O}\big((c_I n^2 + c_R n + c_M n + c_O n + D_M n^2 + n^2)\kappa\big)$ bits, where D_M denotes the multiplicative depth of the circuit. The protocol is perfectly secure against an adaptive active adversary corrupting $t < n/3$ players.*

If all players follow the protocol then PreparationPhase succeeds (and all honest players agree on that). If the PreparationPhase succeeds it generates correct multiplication triples unknown to the adversary and thus the invoked ComputationPhase robustly, correctly, and privately evaluates the circuit.

Note that (as the computation phase is robust given that the preparation phase succeeded) by making the preparation phase robust we get a robust protocol for reactive MPC.

In the following sections we present Player Elimination [HMP00] – a general technique for transforming non-robust protocols into robust protocols at essentially no additional costs.

4.11 Player-Elimination

Player Elimination (PE) is a general technique (from [HMP00]), used for constructing efficient MPC protocols. It allows to transform (typically very efficient) non-robust protocols into robust protocols at essentially no additional costs.

The basic idea is to divide the computation into segments and repeat the non-robust evaluation of each segment until it succeeds, whereby limiting the total number of times the adversary can cause a segment to fail. Each evaluation of a segment proceeds in three steps: (1.) detectable computation (2.) fault detection and (3.) fault localization.

In the detectable computation, the actual non-robust (but detectable) protocol is invoked to compute the segment. In the fault detection the players agree on whether or not there are some unhappy players. If all players are happy, the computation of the segment was successful, the players keep the output and proceed to the next segment. Otherwise the segment failed, the output is discarded and a pair of players $E = \{P_i, P_j\}$

containing at least one corrupted player is localized in the fault localization, eliminated from the actual player set and the segment is repeated with the new player set.[14] We denote the original player set as $\mathcal{P}$ (containing n players, up to t of them faulty), and the actual (reduced) player set as $\mathcal{P}'$ (containing n' players, up to t' of them faulty).

By selecting the size of a segment such that there are t segments, the overall costs of the resulting robust protocol are at most twice the costs of the non-robust protocol (plus the overhead costs for the fault detection and the player elimination).

Technically, a player-elimination protocol proceeds as follows:

Protocol with Player-Elimination.

Let $\mathcal{P}' \leftarrow \mathcal{P}$, $n' \leftarrow n$, $t' \leftarrow t$. Divide computation into t segments of similar size, and do the following for each segment:

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to "happy" (i.e., P_i did not observe a fault).
1. DETECTABLE COMPUTATION: Compute the actual segment in detectable manner, such that (i) if all players in $\mathcal{P}'$ follow their protocol, then the computation succeeds and all players remain happy, and (ii) if the output is incorrect, then at least one honest player in $\mathcal{P}'$ detects so and gets unhappy.
2. FAULT DETECTION: Reach agreement on whether or not all players in $\mathcal{P}'$ are happy (involves Byzantine Agreement). If all players are happy, proceed with the next segment. If at least one player is unhappy, proceed with the following fault-localization procedure.
3. FAULT LOCALIZATION: Find $E \subseteq \mathcal{P}'$ with $|E| = 2$, containing at least one corrupted player.
4. PLAYER ELIMINATION: Set $\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $n' \leftarrow n' - 2$, $t' \leftarrow t' - 1$, and repeat the segment.

[14] Note that we eliminate *players* and not *users*. If a party playing the role of a player as well as the role of a user is eliminated from the player set, it still keeps its user role – can give input and receive output.

4.12 The Robust MPC Protocol

When applying the player-elimination technique to a concrete detectable protocol, two main issues have to be considered: privacy in the fault localization and the shrinking player set.

In our setting, privacy in the fault localization is no problem. We use PE in the preparation phase only, where we work with independent random values only. Thus, in case of faults, everything from the actual segment can be discarded and so privacy is of no concern.

However, special care needs to be taken such that the computation after a (sequence of) player elimination is "compatible" with the outputs of previous segments. We ensure this compatibility be fixing the degree of all sharings to t, independent of the actual threshold t'.

Note that a sharing (among $\mathcal{P}'$) of degree t can be robustly reconstructed as long as $t + 2t' < n'$, what is clearly satisfied when $t < n/3$. Thus the circuit can be robustly evaluated by the players in $\mathcal{P}'$ invoking the old protocol ComputationPhase for the actual player set $\mathcal{P}'$. However, the old protocol GenerateTriples from the preparation phase does not necessarily work for $\mathcal{P}' \subseteq \mathcal{P}$: In order to detectably reconstruct a $2t$-sharing (in the multiplication protocol) $2t + t' < n'$ is required, which is not necessarily satisfied in $\mathcal{P}'$. Thus in order to use PE in the preparation phase, the protocol GenerateTriples has to be adjusted to be able to generate multiplication triples t-shared among the players in $\mathcal{P}'$.

4.12.1 Protocol Overview

The (robust) actively secure protocol consists of a preparation phase and a computation phase (both robust).

In the preparation phase a bunch of correct t-shared secret random triples is generated. The preparation phase uses player elimination, i.e. the triples are generated in a non-robust computation and every time the computation fails a pair of players is localized and eliminated from the player set. Thus the generated triples are shared among the players in the actual player set $\mathcal{P}' \subseteq \mathcal{P}$ (whereby after every elimination holds $t < n' - 2t'$).

After the preparation phase the users are informed about the actual player set $\mathcal{P}'$.

The computation phase is very similar to the one of NonRobusteMPC (Section 4.10.3) – the gates of the circuit are robustly evaluated with the help of the pre-generated t-sharings of random triples. However, as the triples from the preparation phase are t-shared among the players in $\mathcal{P}'$ rather than $\mathcal{P}$, the circuit is evaluated by the players in $\mathcal{P}'$ (whereby all intermediate values are t-shared among the players in $\mathcal{P}'$). The computation phase is still robust, as $t < n' - 2t'$, which is enough for robust reconstruction of a t-sharing in $\mathcal{P}'$.

4.12.2 Preparation Phase

The goal of the preparation phase is to generate correct t-sharings of $c_M + c_R + c_I$ secret random triples (a_k, b_k, c_k), such that $c_k = a_k b_k$ for $k = 1, \ldots, c_M + c_R + c_I$. We stress that all resulting sharings must be t-sharings (rather than t'-sharings) among the player set $\mathcal{P}'$.

Remember, that the protocol GenerateTriples(t) cannot be ran in $\mathcal{P}'$ (with n' players and threshold t'), as it would require $2t < n' - t'$. This is due to the reconstruction of a $2t$-sharing for the degree reduction in the multiplication protocol.[15] Thus we adjust GenerateTriples for the reduced player set $\mathcal{P}'$ by computing $2t'$-sharings (instead of $2t$-sharings) of the products and reducing them to t-sharings..

The new protocol GenerateTriples′ allows (if successful) the players in $\mathcal{P}'$ to generate random multiplication triples t-shared among the players in $\mathcal{P}'$. The protocol GenerateTriples′ (and all invoked sub-protocols) is run in the actual player set $\mathcal{P}'$ with n' players and threshold t'.

The idea of GenerateTriples′ is the following: First DoubleShareRandom is invoked 3 times to generated the random double-sharings $[a_1]_{t,t'}, \ldots, [a_T]_{t,t'}$, $[b_1]_{t,t'}, \ldots, [b_T]_{t,t'}$, and $[r_1]_{t,2t'}, \ldots, [r_T]_{t,2t'}$, respectively. Then for every pair a_k, b_k, a t-sharing of the product $c_k = a_k b_k$ is computed by reducing the locally computed $2t'$-sharing $[c_k]_{2t'} = [a_k]_{t'}[b_k]_{t'}$ to a t-sharing $[c_k]_t$ using the t-sharing $[r_k]_t$ and the $2t'$-sharing $[r_k]_{2t'}$ of the random value r_k.

Protocol GenerateTriples′**.**

[15] Remember that in order to detectably reconstruct a correct d-sharing in $\mathcal{P}'$, it must hold $d < n' - t'$.

1. GENERATE RANDOM DOUBLE SHARINGS $[a]_{t',t}$ AND $[b]_{t',t}$: Invoke DoubleShareRandom(t, t') two times in parallel to generate T random (t, t')-double-sharings $[a_1]_{t,t'}, \ldots, [a_T]_{t,t'}$ and T random (t, t')-double-sharings $[b_1]_{t,t'}, \ldots, [b_T]_{t,t'}$.
2. GENERATE RANDOM DOUBLE SHARINGS $[r]_{t,2t'}$: Invoke DoubleShareRandom$(t, 2t')$ to generate T random double sharings $[r_1]_{t,2t'}, \ldots, [r_T]_{t,2t'}$
3. COMPUTE t-SHARING OF $c = ab$ USING $[r]_{t,2t'}$:
   - 3.1 COMPUTE $[c]_{2t'}$: For $k = 1, \ldots T$ the players compute (locally) the $2t'$-sharing $[c_k]_{2t'}$ of $c_k = a_k b_k$ as $[c_k]_{2t'} = [a_k]_{t'}[b_k]_{t'}$ (by every player computing the product of his shares).
   - 3.2. COMPUTE $[c]_{2t'} \to [c]_t$:
     - 3.2.1 For $k = 1, \ldots T$ the players compute (locally) the $2t'$-sharing of the difference $\delta_k = c_k - r_k$ as $[\delta_k]_{2t'} = [c_k]_{2t'} - [r_k]_{2t'}$.
     - 3.2.2 Invoke ReconsPubl$([\delta_1]_{2t'}, \ldots, [\delta_T]_{2t'}$ to reconstruct the differences $\delta_1, \ldots, \delta_T$ towards every player in $\mathcal{P}'$.
     - 3.2.3 For $k = 1, \ldots T$ the players compute (locally) the t-sharing $[c_k]_t = [r_k]_t + \delta_k$.
4. OUTPUT: The players output the t-sharings of the T triples $([a_1]_t, [b_1]_t, [c_1]_t), \ldots, ([a_T]_t, [b_T]_t, [c_T]_t)$.

Lemma 10 *If* GenerateTriples′ *succeeds (i.e., all honest players are happy), it outputs independent random t-sharings of $T = n - 2t$ random triples $(a_1, b_1, c_1), \ldots, (a_T, b_T, c_T)$ with a_k, b_k independent uniform random values and $c_k = a_k b_k$ for $k = 1, \ldots, T$.* GenerateTriples′ *communicates $\mathcal{O}(n^2\kappa)$ bits.*

Now we can describe the preparation phase.

The following protocol PreparationPhase divides the generation of the $c_M + c_R + c_I$ triples into t segments of length $\ell = \lceil \frac{c_M+c_R+c_I}{t} \rceil$. In each segment the triples are generated invoking the non-robust protocol GenerateTriples′ (as often as necessary), then the players reach agreement on whether or not all players are happy. If yes, they proceed to the next segment. Otherwise, a pair of players is identified in FaultLocalization, excluded from the actual player set $\mathcal{P}'$ and the segment is repeated (with the new $\mathcal{P}'$ and all players setting their happy-bit to "happy").

Protocol PreparationPhase**.**

For each segment $k = 1, \ldots, t$ do:

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to "happy".
1. TRIPLE GENERATION: Invoke $\mathsf{GenerateTriples}'$ $\lceil \frac{\ell}{T} \rceil$ times in parallel.
2. FAULT DETECTION: Invoke $\mathsf{FaultDetection}$ to reach agreement on whether or not at least one player is unhappy. If the output is "succeeded", then the generated triples are outputted and the segment is finished. Otherwise the following Fault-Localization step is executed.
3. FAULT LOCALIZATION: Localize $E \subseteq \mathcal{P}'$ with $|E| = 2$ and at least one player in E being corrupted:
   - 3.0 Denote the player $P_r \in \mathcal{P}'$ with the smallest index r as the referee.[16]
   - 3.1 Every $P_i \in \mathcal{P}'$ sends all values he received and all random values he chose during the computation of the actual segment (including fault detection) to P_r.
   - 3.2 Given the values received in Step 3.1, P_r can reproduce every message that should have been sent (by applying the respective protocol instructions of the sender), and compare it with the value that the recipient claims to have received. Then P_r broadcasts (l, i, j, x, x'), where l is the index of a message where P_i should have sent x to P_j, but P_j claims to have received $x' \neq x$.
   - 3.3 The accused players broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$, if P_j disagrees, set $E = \{P_r, P_j\}$, otherwise set $E = \{P_i, P_j\}$.
4. PLAYER ELIMINATION: Set $\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $n' \leftarrow n' - 2$, $t' \leftarrow t' - 1$, and repeat the segment.

Lemma 11 *The protocol* $\mathsf{PreparationPhase}$ *robustly generates independent random t-sharings of $c_M + c_R + c_I$ secret triples (a_k, b_k, c_k) with a_k, b_k independent uniform random values and $c_k = a_k b_k$ for $k = 1, \ldots, c_M + c_R + c_I$. The generated values are t-shared among the players in the actual player set $\mathcal{P}'$ with n' players (up to t' of them corrupted) and it holds $t < n' - 2t'$.*

$\mathsf{PreparationPhase}$ *communicates $\mathcal{O}\big((c_M + c_R + c_I)n\kappa + n^2\kappa + t\,\mathcal{BA}(\kappa)\big)$ bits, which amounts to $\mathcal{O}\big((c_M + c_R + c_I)n\kappa + n^3\kappa\big)$ bits overall.*

[16]The communication can be balanced by selecting a player who has not yet been referee in a previous segment.

4.12.3 Computation Phase

The computation phase is almost identical to the (robust) computation phase of the non-robust MPC protocol from Section 4.10.3. The only difference is that the circuit is evaluated by the players in the actual player set $\mathcal{P}'$ (instead of $\mathcal{P}$). Thus all invoked sub-protocols are invoked for the player set $\mathcal{P}'$, with n' players, up to $t' < n'/3$ of them corrupted.

In the computation phase, the circuit is robustly evaluated, whereby all intermediate values are t-shared among the players in $\mathcal{P}'$.

Protocol ComputationPhase**.**

Evaluate the gates of the circuit as follows:

- Input Gate (User U inputs s): Let $[r]_t$ denote the associated random sharing. The protocol $\mathsf{VShare}(U, t, s, [r]_t)$ is invoked to let the user U verifiably t-share his input s.
- Addition/Linear Gate: Every $P_i \in \mathcal{P}$ applies the linear function on his respective shares.
- Random Gate: Pick the sharing $[r]_t$ associated with the gate.
- Multiplication Gate: Up to $\lfloor T/2 \rfloor$ multiplication gates are processed in parallel by invoking the protocol $\mathsf{MultiplicationGate}(t)$ (using the associated multiplication triples $([a_1]_t, [b_1]_t, [c_1]_t), \ldots, ([a_{T/2}]_t, [b_{T/2}]_t, [c_{T/2}]_t)$).
- Output Gate (output $[s]$ to User U): Invoke $\mathsf{ReconsPriv}(U, t, [s]_t)$.

Lemma 12 *The protocol* ComputationPhase *robustly evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, given $c_I + c_R + c_M$ pre-shared random multiplication triples, with communicating $\mathcal{O}\big((c_I n + c_M n + c_O n + D_M n^2)\kappa + c_I\, \mathcal{BA}(\kappa)\big)$ bits, where D_M denotes the multiplicative depth of the circuit.*

4.12.4 The Robust Actively Secure MPC Protocol

The actively secure (robust) MPC protocol tolerating up to $t < n/3$ corrupted players consists of two robust phases — the preparation phase and the computation phase.

The preparation phase generates a bunch of t-shared multiplication triples (a, b, c). Remember that preparation phase makes use of player elimination, thus the generated triples are shared among the players in the *actual* player set $\mathcal{P}'$ (with n' players up to t' of them corrupted), where $\mathcal{P}' \subset \mathcal{P}$.

At the end of the preparation phase all users are informed about the actual player set $\mathcal{P}'$.

Then the computation phase is started and the circuit is robustly evaluated by the players in the actual player set $\mathcal{P}'$ using the pre-generated t-shared triples.

Protocol Main**.**

1. PREPARATION PHASE: Set $\mathcal{P}' = \mathcal{P}, n' = n, t' = t$ and invoke PreparationPhase.
2. PROPAGATE $\mathcal{P}'$: Every player in $\mathcal{P}'$ sends to every user the set $\mathcal{P}'$. The user accepts the set he received at least $t+1$ times.
3. COMPUTATION PHASE: Invoke ComputationPhase for the player set $\mathcal{P}'$.

Theorem 4 *The MPC protocol* Main *evaluates a circuit with c_I input, c_R random, c_M multiplication, and c_O output gates, with communicating $\mathcal{O}\big((c_I n + c_R n + c_M n + c_O n + D_M n^2)\kappa + (c_I + n)\,\mathcal{BA}(\kappa)\big)$ bits, which amounts to $\mathcal{O}\big((c_I n^2 + c_R n + c_M n + c_O n + D_M n^2)\kappa + n^3\kappa\big)$ bits, where D_M denotes the multiplicative depth of the circuit. The protocol is perfectly secure against an active adaptive adversary corrupting $t < n/3$ players.*

The amortized communication complexity for giving input can be improved from $\mathcal{O}(n^2\kappa)$ per input to $\mathcal{O}(n\kappa)$. Details can be found in [BH08].

Chapter 5

Active MPC with Setup

5.1 Introduction

In this chapter we present an unconditionally secure MPC protocol secure against an active adversary corrupting up to $t < n/2$ of the players communicating $\mathcal{O}(n^2\kappa)$ bits per multiplication.

The communication complexity of our protocol is to be compared with the most efficient previously known protocol for the same model, which requires *broadcasting* $\Omega(n^5)$ field elements per multiplication.

This substantial reduction in communication is mainly achieved by applying a new technique called *dispute control*: During the course of the protocol, the players keep track of disputes that arise among them, and the ongoing computation is adjusted such that known disputes cannot arise again. Dispute control is inspired by the player-elimination framework. However, player elimination is not suited for models with $t \geq n/3$.[17]

The work presented in this chapter was published in [BH06].

[17]For example, for $n = 2t+1$ could the number of honest players after one elimination be equal to the number of corrupted players before the elimination. Thus after the elimination the remaining honest players would not have any joint information on any intermediate shared value.

5.2 Model

We consider a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, which are connected with a complete network of secure synchronous channels.[18] Furthermore, we assume the availability of broadcast channels. These can be simulated when a trusted setup is available [PW92].

The adversary corrupts up to t players for any fixed t with $t < n/2$. The adversary is computationally unbounded, active, adaptive and rushing.

The security of our protocols is information-theoretic with a negligible error probability of $2^{-\mathcal{O}(\kappa)}$ for some security parameter κ.

The function to be computed is specified as an arithmetic circuit over a finite field $\mathbb{F} = \mathrm{GF}(2^\kappa)$, with input, addition, multiplication, random, and output gates. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O.

To every player $P_i \in \mathcal{P}$, a unique non-zero element $\alpha_i \in \mathbb{F} \setminus \{0\}$ is assigned.

5.3 Dispute Control

In the active model, the adversary can provoke inconsistencies among the honest players, who therefore regularly have to check their views and, in case of inconsistencies, invoke some fault-recovery procedure. These checks tend to be very expensive (they require invocations to a Byzantine agreement primitive), and must be performed even when no player deviates from the protocol.

The goal of *dispute control* is to reduce the frequency of faults by publicly identifying (localizing) a pair of disputing players (at least one of them corrupted) whenever a fault is observed and preventing this pair from getting into dispute ever again. Hence, the number of faults that can occur during the whole protocol is limited to $t(t+1)$.

The localized disputes are filed in a publicly known *dispute set* $\Delta \subseteq \mathcal{P} \times \mathcal{P}$, a set of unordered pairs of players that are in dispute with each other. A pair $\{P_i, P_j\} \in \Delta$ means that there is a dispute between P_i

[18]For the sake of simplicity, we assume that the set of users is identical to the set of players.

and P_j, hence either P_i or P_j (or both) are corrupted. Note that from the point of view of P_i, the players $\{P_j \mid \{P_i, P_j\} \in \Delta\}$ are corrupted, and P_i doesn't care for them; in particular, he won't send or receive any private messages from them. As no honest player can be in dispute with more than t players, we automatically include the pairs $\{P_i, P_j\}$ for every $P_j \in \mathcal{P}$ once P_i is involved in more than t disputes. Furthermore, we define the set $\mathcal{X}$ to be the set of players who are undoubtedly detected to be corrupted, i.e., those players who are in dispute with more than t other players.

Once dispute control is in place, we can take advantage of the fact that the number of faults during the protocol is limited and reduce the number of expensive consistency checks: We divide the protocol into n^2 *segments*, run each segment without any consistency checks and only at the end of the segment check all operations of the segment in a single verification step. If the verification fails, a new dispute is localized, and the segment is repeated. At most $t(t+1)$ segments can fail, and the total number of segment evaluations (including repetitions) is at most $n^2 + t(t+1)$, hence the overhead for repeating failed segments is only a factor of 2. Formally the evaluation of each segment proceeds as follows:

1. **Private (dispute-aware) computation.** The effective protocol is computed very efficiently but non-robustly. This computation is adjusted to prevent faults due to disputes that are already registered in the dispute set Δ. In particular, players in dispute do not communicate with each other privately.

2. **Fault detection.** The players jointly find out whether or not a fault has occurred. This step typically requires each player to broadcast one bit indicating whether or not he observed an inconsistency within the current segment. If no fault is reported, then the computation of the segment is completed, and the next segment is evaluated. If at least one fault is reported, we say that the segment has failed, and the following step is performed.

3. **Fault localization and dispute control.** The players publicly identify a pair $\{P_i, P_j\}$ of players, where at least one of them is corrupted and has deviated from the protocol, and who are not yet registered in Δ. Then we set $\Delta \leftarrow \Delta \cup \{P_i, P_j\}$ and restart the current segment.

5.4 Three-Level Secret-Sharing

We use three different levels of secret-shadings, all based on Shamir's sharing [Sha79], ameliorated with dispute control. The weakest level, called *1D-sharing*, is a polynomial sharing scheme, where the players who are in dispute with the dealer (implicitly) receive a fixed-0 share, called *Kudzu-share*. In order to *1D-share* a value s, the dealer P_D selects a random degree-t polynomial $f(x)$ with $f(0) = s$ and $f(\alpha_i) = 0$ for every $\{P_D, P_i\} \in \Delta$, and sends the shares $s_i = f(\alpha_i)$ to every $P_i \in \mathcal{P}$ (the Kudzu-shares are not really sent; instead, the receiver sets his share to 0). A protocol VSS1D for verifiably 1D-share a bunch of values will be given in Section 5.6.2. Note that 1D-sharings are not robust; reconstruction requires that all players (except those with Kudzu-shares) cooperate. However, they are detectable in the sense that it can be decided whether or not the reconstruction was successful.

The middle level of secret sharing, called *2D-sharing*, is a two-level polynomial sharings scheme: The share s_i of each player $P_i \in \mathcal{P}$ is 1D-shared among the players (for dealer P_i). More precisely, a value s is 2D-shared when there exists degree-t polynomials $f, f_1, \ldots, f_n$ with $f(0) = s$ and, for $i = 1, \ldots, n$, $f_i(0) = f(\alpha_i)$ and $\forall P_j \in \mathcal{P} : \{P_i, P_j\} \in \Delta \rightarrow f_i(\alpha_j) = 0$. Every player $P_i \in \mathcal{P}$ holds a share $s_i = f(\alpha_i)$ of s, the polynomial $f_i(x)$ for sharing s_i, and a share-share $s_{ji} = f_j(\alpha_i)$ of the share s_j of every player $P_j \in \mathcal{P}$. We say that P_i *owns* the 1D-sharing of s_i, which means in particular that players who are in dispute with P_i hold 0 as share-share of s_i. We will never have a dealer 2D-share a value; instead, we will upgrade 1D-sharings (or rather sums of 1D-sharings) to 2D-sharings, using protocol Upgrade1Dto2D. Note that also 2D-sharings are not robust.

The strongest level of secret sharing, called *2D*-sharing*, is a 2D-sharing, where in addition, the share-shares are secured with information checking (see Section 5.6.5). More precisely, for each share-share s_{ij} (which is not a Kudzu-share, i.e., $\{P_i, P_j\} \notin \Delta$), the owner P_i of the sharing has provided authentication tags for every verifier $P_V \in \mathcal{P}$ who is neither in dispute with the owner P_i nor the recipient P_j, i.e., $\{P_V, P_i\} \notin \Delta$ and $\{P_V, P_j\} \notin \Delta$. These authentication tags allow P_V in the reconstruction to verify the correctness of the received share-shares; hence, 2D*-sharings are robust. Actually, P_i does not distribute authentication tags for every single share-share s_{ij}, but rather for huge collections of many share-shares $s_{ij}^{(1)}, \ldots, s_{ij}^{(\ell)}$, and P_V can only verify the correctness of all share-shares at once. Also 2D*-sharings are never distributed by a dealer;

instead, we will upgrade collections of 2D-sharings to 2D*-sharings, using protocol Upgrade2Dto2D*.

5.5 Main Protocol — Overview

The main protocol proceeds in three phases (each making use of segmentation and dispute control):

Preparation phase: The preparation phase uses the circuit-randomization technique of Beaver [Bea91a]: A number of so-called multiplication triples (a, b, c) with $c = ab$ are generated and shared among the players. These triples will then be used in the computation phase for efficiently multiplying shared values. Furthermore, a number of random values are generated and shared, which will be used as outputs of random gates.

Input phase: In the input phase, every player with input shares his input among the players.

Computation phase: In the computation phase, the circuit is evaluated gate by gate (level by level), with help of the prepared multiplication triples and the random values. Given the sharings of the multiplication triples, the random values, and the inputs, the computation phase is fully deterministic. Indeed, the computation phase can be seen as a sequence of reconstructions of known linear combinations of shared values.

Each phase uses dispute control. We initialize the dispute set $\Delta = \{\}$ and enter the first segment of the preparation phase. Then we evaluate segment by segment, and with each segment that fails and is to be repeated, the dispute set Δ grows. Once all segments of the preparation phase have succeeded, the players move on to the first segment of the input phase. Also in this phase, segments can fail and have to be repeated. This allows corrupted players to change their inputs. However, as the adversary obtains no information about whatsoever in the input phase, this does not affect the independence of the inputs. Once all input segments have succeeded, the players move on to the first segment of the computation phase. In this phase, the players (and hence also the adversary) do obtain information about their outputs; however, the computation stage is fully deterministic. Even when a segment fails (and is repeated) *after* the adversary has learned some output, he cannot influence the outputs of the honest players anymore.

In the preparation phase and in the input phase, the private computation is highly parallelized. All proposed sub-protocols process many inputs at once, producing many outputs. This helps reducing the costs for the fault detection and localization, as for all parallel instances, only one single fault-handling procedure is executed. Often, instead of verifying single instances of some test data, we will verify a random linear combination of many instances. Note that the protocols themselves do not use broadcast, but fault handling does.

5.6 Sub-Protocols

All sub-protocols have a private (dispute aware) computation, a fault detection and a fault localization. They can succeed or fail and the players always agree (using broadcast in the fault detection) on what is the case. In case of a failure the public output of the sub-protocol is a (new) pair of players $E = \{P_i, P_j\} \notin \Delta$ such as either P_i or P_j (or both) are corrupted. If some invoked sub-protocol fails with $E = \{P_i, P_j\}$ then the invoking sub-protocol fails with $E = \{P_i, P_j\}$ and is aborted (this abort will be handled in the main protocol).

5.6.1 Dispute-Control Broadcast

The protocol DC-Broadcast allows every sender $P_S \in \mathcal{P} \setminus \mathcal{X}$ to distribute a vector of ℓ values $s^{(1,S)}, \ldots, s^{(\ell,S)}$ among the players in $\mathcal{P} \setminus \mathcal{X}$, such that it is guaranteed that all honest recipients receive the same vectors (or the protocol fails).

This protocol is rather simple: Every sender directly transmits his vector to the players he is not in dispute with, and via another player to those players he is in dispute with. Then the players pair-wisely compare their vectors by using universal hash functions [CW79]. As universal hash with key $k \in \mathbb{F}$, we use the function $U_k : \mathbb{F}^\ell \to \mathbb{F}, (s^{(1)}, \ldots, s^{(\ell)}) \mapsto s^{(1)} + s^{(2)}k + \ldots + s^{(\ell)}k^{\ell-1}$. The probability that two different vectors map to the same hash value for a uniformly chosen key is at most $\ell/|\mathbb{F}|$, which is negligible in our setting with $\mathbb{F} = \mathrm{GF}(2^\kappa)$.

Protocol DC-Broadcast**.**

1. Private Computation: The following steps are executed in parallel for every sender $P_S \in \mathcal{P} \setminus \mathcal{X}$:
   1.1 P_S sends $s^{(1,S)}, \ldots, s^{(\ell,S)}$ to every P_i with $\{P_S, P_i\} \notin \Delta$.
   1.2 For every P_i with $\{P_S, P_i\} \in \Delta$ (but $P_i \notin \mathcal{X}$), the smallest player $P_{i'}$ with $\{P_S, P_{i'}\} \notin \Delta$ and $\{P_{i'}, P_i\} \notin \Delta$ forwards $s^{(1,S)}, \ldots, s^{(\ell,S)}$ to P_i.[19] We call $P_{i'}$ the *proxy* of P_i.
2. Fault Detection: The following steps are executed in parallel for every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$:
   2.1 P_V selects a key $k_V \in_R \mathbb{F}$ for a universal hash function U_k and sends it to every P_i with $\{P_V, P_i\} \notin \Delta$.
   2.2 Every P_i with $\{P_V, P_i\} \notin \Delta$ sends the values $h_{S,i} = U_{k_V}(s^{(1,S)}, \ldots, s^{(\ell,S)})$ for every P_S to P_V.
   2.3 P_V broadcasts a bit "accept" or "reject", indicating whether for every $P_S \in \mathcal{P} \setminus \mathcal{X}$, the hash values $h_{S,i}$ of each P_i with $\{P_V, P_i\} \notin \Delta$ are equal.

   If every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$ broadcasts "accept" in Step 2.3, then the protocol succeeds and terminates.
3. Fault Localization: The following steps are executed for the smallest $P_V \in \mathcal{P} \setminus \mathcal{X}$ reporting a fault.
   3.1 P_V selects S, i, j such that $P_S \notin \mathcal{X}$, $\{P_V, P_i\} \notin \Delta$, and $\{P_V, P_j\} \notin \Delta$, and $h_{S,i} \neq h_{S,j}$, and broadcasts $S, i, j, h_{S,i}, h_{S,j}$, and $k = k_V$.
   3.2 We denote the proxies of P_i and P_j by $P_{i'}$ and $P_{j'}$, respectively (if no proxy exists, we set $i' = i$, respectively $j' = j$). The players $P_S, P_i, P_j, P_{i'}, P_{j'}$ all compute and broadcast a hash value with key k of their vector $s^{(1,S)}, \ldots, s^{(\ell,S)}$, denoted as $h_S, h_i, h_j, h_{i'}, h_{j'}$, respectively. The protocol fails with E being the first pair (P_V, P_i), $(P_i, P_{i'})$, $(P_{i'}, P_S)$, $(P_S, P_{j'})$, $(P_{j'}, P_j)$, or (P_j, P_V), where $h_{S,i} \neq h_i$, $h_i \neq h_{i'}$, $h_{i'} \neq h_S$, $h_S \neq h_{j'}$, $h_{j'} \neq h_j$, or $h_j \neq h_{S,j}$, respectively.

Lemma 13 *If* DC-Broadcast *succeeds, then with overwhelming probability, for each sender $P_S \in \mathcal{P} \setminus \mathcal{X}$, all honest players in $\mathcal{P}$ hold the same vector $s^{(1,S)}, \ldots, s^{(\ell,S)}$, which is the vector of P_S if honest. If the protocol fails, a*

[19]The existence of such a player $P_{i'}$ for $P_{S'} \notin \mathcal{X}$ and $P_i \notin \mathcal{X}$ follows by a counting argument.

new dispute pair E is localized. The protocol communicates $\mathcal{O}(\ell n^2 + n^3)$ and broadcasts $\mathcal{O}(n)$ field elements.

Proof: In order to prove that all honest players output the same vector $s^{(1,S)}, \ldots, s^{(\ell,S)}$ when the protocol succeeds, consider two honest players P_i and P_j. As both P_i and P_j are honest, $\{P_i, P_j\} \notin \Delta$ holds, and P_i and P_j have mutually exchanged universal hash values in Step 2. Hence, with overwhelming probability, a difference in the vectors would have been detected and the protocol would have failed. It follows immediately from the protocol that when P_S is honest and the protocol succeeds, then all honest players receive the vector directly from P_S. When the protocol fails with dispute pair E, then one can verify by inspection that the two players in E disagree on a value they have privately exchanged, hence either of the players must be faulty. And as players in dispute do not communicate with each other, the localized dispute pair is new. ■

5.6.2 Verifiable Secret-Sharing

The protocol VSS1D allows every dealer $P_D \in \mathcal{P} \setminus \mathcal{X}$ to verifiably 1D-share ℓ values $s^{(1,D)}, \ldots, s^{(\ell,D)}$ resulting in each player $P_i \in \mathcal{P} \setminus \mathcal{X}$ holding the shares $s_i^{(1,D)}, \ldots, s_i^{(\ell,D)}$ for each dealer P_D. The correctness of these sharings is verified by letting every player take on the role of a verifier P_V and inspect a random linear combination of the sharings of each dealer P_D. For privacy reasons, each such random linear combination is blinded with a random 1D-sharing, i.e., every dealer P_D 1D-shares additional n blinding values $s^{(\ell+1,D)}, \ldots, s^{(\ell+n,D)}$.

Protocol VSS1D**.**

1. PRIVATE COMPUTATION: Every dealer $P_D \in \mathcal{P} \setminus \mathcal{X}$ selects n random blindings $s^{(\ell+1,D)}, \ldots, s^{(\ell+n,D)}$ (one for each verifier). Then, P_D 1D-shares $s^{(1,D)}, \ldots, s^{(\ell+n,D)}$, i.e., for every $m = 1, \ldots, \ell+n$, P_D picks a random degree-t polynomial $f^{(m,D)}(x)$ with $f^{(m,D)}(0) = s^{(m,D)}$ and $f^{(m,D)}(\alpha_i) = 0$ for every i with $\{P_D, P_i\} \in \Delta$ (the Kudzu-shares), and sends the share $s_i^{(m,D)} = f^{(m,D)}(\alpha_i)$ to every player P_i with $\{P_D, P_i\} \notin \Delta$; every player P_i with $\{P_D, P_i\} \in \Delta$ sets his share $s_i^{(m,D)} = 0$.

2. FAULT DETECTION: Every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$ selects a random challenge vector $(r^{(1,V)}, \ldots, r^{(\ell,V)})$. Then, DC-Broadcast is invoked to let every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$ distribute his vector among the players $P_i \in \mathcal{P} \setminus \mathcal{X}$. Then the following steps are executed for every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$ (we suppress the index V and denote the challenge vector $(r^{(1)}, \ldots, r^{(\ell)})$):
   2.1 For every dealer P_D, the random linear combination $f^{(*,D)}(x)$ of his 1D-sharings is defined as $f^{(*,D)}(x) = \sum_{m=1}^{\ell} r^{(m)} f^{(m,D)}(x) + f^{(\ell+V,D)}(x)$. Accordingly, for every dealer P_D, every player P_i with $\{P_i, P_D\} \notin \Delta$ and $\{P_i, P_V\} \notin \Delta$ sends to P_V his share $s_i^{(*,D)}$ on $f^{(*,D)}(x)$, i.e., $s_i^{(*,D)} = \sum_{m=1}^{\ell} r^{(m)} s_i^{(m,D)} + s_i^{(\ell+V,D)}$.
   2.2 For each dealer $P_D \in \mathcal{P} \setminus \mathcal{X}$, the verifier P_V checks whether the received shares $s_i^{(*,D)}$ define a correct 1D-sharing for P_D, i.e., whether there exists a degree-t polynomial $\widetilde{f}^{(*,D)}(x)$ with $\widetilde{f}^{(*,D)}(\alpha_i) = s_i^{(*,D)}$ for every i with $\{P_V, P_i\} \notin \Delta$ and $\{P_D, P_i\} \notin \Delta$, and $\widetilde{f}^{(*,D)}(\alpha_i) = 0$ for every i with $\{P_D, P_i\} \in \Delta$ (Kudzu).[20] P_V broadcasts a bit "accept" or "reject", indicating whether or not the the above checks succeed for all dealers.

   If all verifiers $P_V \in \mathcal{P} \setminus \mathcal{X}$ broadcast "accept" the protocol succeeded and terminates.
3. FAULT LOCALIZATION: The following steps are executed for the smallest P_V reporting a fault in Step 2.2.
   3.1 P_V broadcasts the index D of P_D whose polynomial $\widetilde{f}^{(*,D)}(x)$ does not define a correct 1D-sharing.
   3.2 Every player P_i with $\{P_i, P_D\} \notin \Delta$ and $\{P_i, P_V\} \notin \Delta$ broadcasts his share $s_i^{(*,D)}$.
   3.3 If the broadcasted shares define a 1D-sharing for dealer P_D, then P_V broadcasts the index i of a player P_i with $\{P_i, P_V\} \notin \Delta$ and $\{P_i, P_D\} \notin \Delta$ who has broadcasted a different share $s_i^{(*,D)}$ in Step 3.2 than he has privately sent to P_V in Step 2.1, and the protocol fails with $E = \{P_V, P_i\}$. Otherwise, when the broadcasted shares do not define a correct 1D-sharing for dealer P_D, then the dealer broadcasts the index i of a player P_i with $\{P_i, P_D\} \notin \Delta$ who has broadcasted a wrong share $s_i^{(*,D)}$ and the protocol fails with $E = \{P_D, P_i\}$.

[20]Note that any linear combination of Kudzu-shares is Kudzu.

Lemma 14 *If* VSS1D *succeeds, then with overwhelming probability, the values* $s^{(1,D)}, \ldots, s^{(\ell,D)}$ *of each dealer* $P_D \in \mathcal{P} \setminus \mathcal{X}$ *are correctly 1D-shared. If the protocol fails, then the localized pair* $E = \{P_i, P_j\}$ *is new (i.e.,* $E \notin \Delta$*) and either* P_i *or* P_j *(or both) are corrupted. The privacy of the inputs of the honest players is guaranteed through the whole protocol (even if the protocol fails). The protocol communicates* $\mathcal{O}(\ell n^2 + n^3)$ *and broadcasts* $\mathcal{O}(n)$ *field elements.*

Proof: In order to prove the correctness, first consider a dealer P_D, an honest verifier P_V, the (by P_D supposedly correct 1D-shared) values $s^{(1,D)}, \ldots, s^{(\ell,D)}$ and the blinding value $s^{(\ell+V,D)}$. Assume that the sharing of one of the values is not a correct 1D-sharing, i.e., the shares of the honest players (including the Kudzu shares) lie on a polynomial of degree higher than t. Then there are at most $2^{\kappa(\ell-1)}$ (out of $2^{\kappa\ell}$) challenge vectors $(r^{(1)}, \ldots, r^{(\ell)}) \in \mathbb{F}^\ell$ such that the sharing of $s^{(*,D)} = \sum_{m=1}^{\ell} r^{(m)} s^{(m,D)} + s^{(\ell+V,D)}$ is a correct 1D-sharing, i.e. the polynomial defined by the shares of the honest players is of degree t. As the verifier P_V chooses his challenge vector uniformly at random and gets the correctly linearly combined shares from all honest players (an honest verifier is in dispute with no honest player), the probability of him not detecting the fault is at most $2^{\kappa(\ell-1)}/2^{\kappa\ell} = 1/2^\kappa$. Thus the probability that the protocol succeeds in case of at least one faulty sharing (from any dealer) is negligible.

The privacy of the inputs of the honest players follows from the fact that up to t shares give no information about the secret and from the fact that the reconstructed linear combinations are blinded with a random value chosen by the dealer himself (for every verifier a different one) and are so (for every honest dealer) statistically independent from the dealers secret.

If the protocol fails, then the localized dispute pair consists of two players who have publicly disagreed on a value they have privately exchanged in some previous step (or a value computed from such values), therefore it is obvious, that at least one of them is corrupted. As only players who are not in dispute with each other communicate privately, the localized dispute is a new one. ∎

We present a protocol for reconstructing sums of correct 1D-sharings. Consider a set $\mathcal{P}_D \subseteq \mathcal{P} \setminus \mathcal{X}$ of dealers and a set $\mathcal{P}_R \subseteq \mathcal{P} \setminus \mathcal{X}$ of recipients and the actual dispute set Δ. Every dealer $P_D \in \mathcal{P}_D$ has verifiably 1D-shared (with the actual Δ) ℓ summands $s^{(1,D)}, \ldots, s^{(\ell,D)}$ with the polynomials $f^{(1,D)}(x), \ldots, f^{(\ell,D)}(x)$. We denote the share of $s^{(m,D)}$ for player

$P_i \in \mathcal{P}$ by $s_i^{(m,D)}$, i.e. $s_i^{(m,D)} = f^{(m,D)}(\alpha_i)$. Note that $s_i^{(m,D)} = 0$ when $\{P_D, P_i\} \in \Delta$ (Kudzu). The values $s^{(1)}, \ldots, s^{(\ell)}$ to be reconstructed are defined as the sums of the above summands, i.e., $s^{(m)} = \sum_{P_D \in \mathcal{P}_D} s^{(m,D)}$. Each $s^{(m)}$ is implicitly shared (as Shamir-sharing, not as 1D-sharing) with the polynomial $f^{(m)}(x) = \sum_{P_D \in \mathcal{P}_D} f^{(m,D)}(x)$; we denote the (implicitly defined) share of each player $P_i \in \mathcal{P}$ by $s_i^{(m)}$, i.e. $s_i^{(m)} = f^{(m)}(\alpha_i)$.

Note that in case of a fault, this protocol does not guarantee the privacy of the summands.

Protocol Reconstruct1D.

1. Private Computation: For every $m = 1, \ldots, \ell$, every player $P_i \in \mathcal{P}$ computes his sum share $s_i^{(m)} = \sum_{P_D \in \mathcal{P}_D} s_i^{(m,D)}$, and sends it to every $P_R \in \mathcal{P}_R$ with $\{P_i, P_R\} \notin \Delta$. Every $P_R \in \mathcal{P}_R$ checks for each $m = 1, \ldots, \ell$ whether the received shares lie on a polynomial $\widetilde{f}^{(m)}(x)$ of degree t. If so, it follows that $\widetilde{f}^{(m)}(x) = f^{(m)}(x)$, and P_R reconstructs $s^{(m)} = \widetilde{f}^{(m)}(0)$.
2. Fault Detection: Every $P_R \in \mathcal{P}_R$ broadcasts "accept" or "reject", indicating whether he could reconstruct all values $s^{(m)}$ for $m = 1, \ldots, \ell$ in Step 1. If all recipients broadcast "accept", then the protocol succeeds and terminates.
3. Fault Localization: The following steps are executed for the smallest complaining recipient $P_R \in \mathcal{P}_R$.
   - 3.1 P_R broadcasts the index m of the polynomial $\widetilde{f}^{(m)}(x)$ he could not reconstruct.
   - 3.2 Every player P_i with $\{P_i, P_R\} \notin \Delta$ sends to P_R his summand shares $s_i^{(m,D)}$ for every dealer $P_D \in \mathcal{P}_D$ with $\{P_i, P_D\} \notin \Delta$.
   - 3.3 P_R verifies for every P_i with $\{P_i, P_R\} \notin \Delta$ that the provided summand shares add up to the previously provided sum share, i.e., $\sum_{P_D : \{P_i, P_D\} \notin \Delta} s_i^{(m,D)} = s_i^{(m)}$.[21] In case of a fault, P_R broadcasts the index i of the bad player P_i, and the protocol fails with $E = \{P_i, P_R\}$.
   - 3.4 P_R broadcasts the index D of a dealer $P_D \in \mathcal{P}_D$ such that the received shares $s_i^{(m,D)}$ do not define a correct 1D-sharing for dealer P_D, i.e., there is no degree-t polynomial $f(x)$ with

[21] Note that the Kudzu-shares $s_i^{(*,D)}$ with $\{P_i, P_D\} \in \Delta$ are 0 and do not contribute to the sum.

$f(\alpha_i) = s_i^{(m,D)}$ for every i with $\{P_i, P_R\} \notin \Delta$ and $\{P_i, P_D\} \notin \Delta$, and with $f(\alpha_i) = 0$ (Kudzu) for every i with $\{P_i, P_R\} \notin \Delta$ and $\{P_i, P_D\} \in \Delta$.

3.5 Every player P_i with $\{P_i, P_R\} \notin \Delta$ and $\{P_i, P_D\} \notin \Delta$ broadcasts his summand share $s_i^{(m,D)}$.

3.6 If the broadcasted summand shares define a correct 1D-sharing for dealer P_D, then P_R broadcasts the index i of a player P_i who has broadcasted a different value $s_i^{(m,D)}$ in Step 3.5 than he has privately sent to P_R in Step 3.2, and the protocol fails with $E = \{P_i, P_R\}$. Otherwise, when the broadcasted summand shares do not define a correct 1D-sharing for P_D, then P_D broadcasts the index i of a player P_i who has broadcasted a wrong share $s_i^{(m,D)}$, and the protocol fails with $E = \{P_i, P_D\}$.

Lemma 15 *If the values $s^{(1,D)}, \ldots, s^{(\ell,D)}$ of each $P_D \in \mathcal{P}_D$ are correctly 1D-shared (for the actual Δ), then the following holds: If* Reconstruct1D *succeeds, then the privacy is guaranteed and every value reconstructed towards an honest recipient lies on the degree t polynomial defined by the (at least $t+1$) shares of the honest players. If the protocol fails then the localized pair $E = \{P_i, P_j\}$ is new and contains at least one corrupted player. The protocol communicates $\mathcal{O}(\ell n^2)$ and broadcasts $\mathcal{O}(n)$ field elements.*

Proof: As an honest verifier is not in dispute with any other honest player, he will receive at least $t+1$ shares of the honest players, which uniquely define a degree t polynomial. If the shares received from the corrupted players lie on this polynomial, he will reconstruct the right secret, otherwise the interpolated polynomial will be of degree higher then t and the protocol will fail. The rest follows (along the lines of proof of Lemma 14) from inspection of the protocol. ■

5.6.3 Generating Random Challenges

The following protocol allows the players to generate a publicly known (i.e., to the players in $\mathcal{P} \setminus \mathcal{X}$) challenge vector $s^{(1)}, \ldots, s^{(\ell)}$, or the protocol fails, if one of the sub-protocols fails, and outputs a new dispute pair $E = \{P_i, P_j\}$:

Protocol GenerateChallenges**.**

1. Every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ selects a random summand vector $s^{(1,k)}, \ldots, s^{(\ell,k)}$.
2. Invoke VSS1D to let every P_k verifiably 1D-share his summand vector.
3. Invoke the protocol Reconstruct1D (with $\mathcal{P}_D = \mathcal{P}_R = \mathcal{P} \setminus \mathcal{X}$) to reconstruct the sum sharings $\sum_{P_k \in \mathcal{P}_D} s^{(1,k)}, \ldots, \sum_{P_k \in \mathcal{P}_D} s^{(\ell,k)}$ towards every $P_j \in \mathcal{P}_R$.

Lemma 16 *If* GenerateChallenges *succeeds, then with overwhelming probability, the generated values are uniformly distributed. If the protocol fails, then the localized dispute pair $E = \{P_i, P_j\}$ is new and contains at least one corrupted player. The protocol communicates $\mathcal{O}(\ell n^2 + n^3)$ and broadcasts $\mathcal{O}(n)$ field elements.*

5.6.4 Upgrading 1D-Sharings to 2D-Sharings

We present a protocol for upgrading sums of 1D-sharings to 2D-sharings. The given 1D-sharings must be for the actual Δ; the correctness of these sharings is implicitly verified in the upgrade protocol and must not be a priori guaranteed. The protocol outputs correct 2D-sharings or it fails with a new dispute pair E.

Formally, we consider a set $\mathcal{P}_D \subseteq \mathcal{P} \setminus \mathcal{X}$ of dealers, where each dealer $P_D \in \mathcal{P}_D$ has (for the actual Δ) 1D-shared ℓ summands $s^{(1,D)}, \ldots, s^{(\ell,D)}$ with the polynomials $f^{(1,D)}(x), \ldots, f^{(\ell,D)}(x)$. We denote the share of $s^{(m,D)}$ for player $P_i \in \mathcal{P}$ by $s_i^{(m,D)}$, i.e. $s_i^{(m,D)} = f^{(m,D)}(\alpha_i)$. Note that $s_i^{(m,D)} = 0$ when $\{P_D, P_i\} \in \Delta$ (Kudzu). The values $s^{(1)}, \ldots, s^{(\ell)}$ to be 2D-shared are defined as the sums of the above summands, i.e., $s^{(m)} = \sum_{P_D \in \mathcal{P}_D} s^{(m,D)}$. Each of these values is implicitly shared (as Shamir-sharing, not as 1D-sharing) with the polynomial $f^{(m)}(x) = \sum_{P_D \in \mathcal{P}_D} f^{(m,D)}(x)$; we denote the (implicitly defined) share of each player $P_i \in \mathcal{P}$ by $s_i^{(m)} = f^{(m)}(\alpha_i)$.

Protocol Upgrade1Dto2D**.**

1. Private Computation: The players first jointly generate a sharing of an additional randomly chosen value $s^{(\ell+1)}$. Then, all $\ell + 1$ sharings are upgraded to 2D-sharings, and the correctness is verified with destroying the privacy of this blinding value.

1.1 Every dealer $P_D \in \mathcal{P}_D$ picks a random summand $s^{(\ell+1,D)}$ and 1D-shares it among the players with polynomial $f^{(\ell+1,D)}(x)$, resulting in every player P_i holding a share $s_i^{(\ell+1,D)}$.

1.2 For every $m = 1, \ldots, \ell+1$, every player $P_i \in \mathcal{P} \setminus \mathcal{X}$ computes his share $s_i^{(m)}$ of $s^{(m)} = \sum_{P_D \in \mathcal{P}_D} s^{(m,D)}$ (the value to be 2D-shared) as $s_i^{(m)} = \sum_{P_D \in \mathcal{P}_D} s_i^{(m,D)}$, and 1D-shares it with the polynomial $f_i^{(m)}(x)$, such that $f_i^{(m)}(\alpha_j) = 0$ for $\{P_i, P_j\} \in \Delta$ (Kudzu).[22] We denote the share-shares $f_i^{(m)}(\alpha_j)$ as $s_{ij}^{(m)}$. The 1D-sharing of detected players $P_i \in \mathcal{X}$ is the constant-0 sharing (all share-shares are Kudzu).

2. FAULT DETECTION: In order to verify the correctness of the resulting sharings, the players jointly generate a random challenge vector $(r^{(1)}, \ldots, r^{(\ell)}) \in \mathbb{F}^\ell$ using the protocol GenerateChallenges. Then, the correctness of the 2D-sharing of the random linear combination $\sum_{m=1}^{\ell} r^{(m)} s^{(m)} + s^{(\ell+1)}$ will be verified (in parallel) by every player $P_V \in \mathcal{P} \setminus \mathcal{X}$. We denote the linearly combined polynomials by $f(x) = \sum_{m=1}^{\ell} r^{(m)} f^{(m)}(x) + f^{(\ell+1)}(x)$ (first-level sharing), respectively $f_i(x) = \sum_{m=1}^{\ell} r^{(m)} f_i^{(m)}(x) + f_i^{(\ell+1)}(x)$ (second-level sharings). The following steps are performed in parallel for every verifier $P_V \in \mathcal{P} \setminus \mathcal{X}$:

2.1 Every P_j with $\{P_V, P_j\} \notin \Delta$ computes and sends to P_V the following linear combinations of his share-shares for every $i = 1, \ldots, n$ with $\{P_i, P_j\} \notin \Delta$: $\quad s_{ij} = \sum_{m=1}^{\ell} r^{(m)} s_{ij}^{(m)} + s_{ij}^{(\ell+1)}$.

2.2 P_V checks for each $i = 1, \ldots, n$, whether the received share-shares s_{ij} define a valid 1D-sharing for dealer P_i, i.e., there exists a polynomial $\widetilde{f}_i(x)$ with $\widetilde{f}_i(\alpha_j) = s_{ij}$ for every j with $\{P_V, P_j\} \notin \Delta$ and $\{P_i, P_j\} \notin \Delta$, and $\widetilde{f}_i(\alpha_j) = 0$ (i.e., Kudzu) for every j with $\{P_i, P_j\} \in \Delta$,[23] and broadcasts a bit "accept" or "reject".

2.3. P_V checks that the first-level sharing $\widetilde{f}_1(0), \ldots, \widetilde{f}_n(0)$ is a valid Shamir-sharing of degree t and broadcasts "accept" or "reject".

If all verifiers P_V broadcast "accept" in Steps 2.2 and 2.3, the protocol succeeded and terminates.

[22]Note that the second-level polynomials $f_1^{(m)}(x), \ldots, f_n^{(m)}(x)$ implicitly define a first-level sharing $f_1^{(m)}(0), \ldots, f_n^{(m)}(0)$ of some value. If all players followed the protocol the values $f_1^{(m)}(0), \ldots, f_n^{(m)}(0)$ define a correct Shamir-sharing of $s^{(m)}$.

[23]Observe that in this case $\widetilde{f}_i(x) = f_i(x)$.

3. FAULT LOCALIZATION: The following steps are executed for the smallest complaining verifier P_V.
   3.1 If the reported fault was in Step 2.2, i.e., P_V observed that one of the second-level sharings is not a correct 1D-sharing, the following steps are executed:
      3.1.1 P_V broadcasts the index i of the invalid second-level sharing.
      3.1.2 Every P_j with $\{P_j, P_V\} \notin \Delta$ and $\{P_j, P_i\} \notin \Delta$ broadcasts s_{ij}.
      3.1.3 If the broadcasted shares define a correct 1D-sharing, then there exists a player P_j with $\{P_j, P_V\} \notin \Delta$ who has broadcasted a different value than he has privately sent to P_V in Step 2.1. P_V broadcasts his index j, and the protocol fails with $E = \{P_V, P_j\}$. If the broadcasted shares do not define a correct 1D-sharing, the owner P_i of this second-level sharing broadcasts the index j of a player P_j (with $\{P_i, P_j\} \notin \Delta$) who has broadcasted a wrong share $s_{ij} \neq f_i(\alpha_j)$, and the protocol fails with $E = \{P_i, P_j\}$.
   3.2 If the observed fault was in Step 2.3, i.e., P_V could correctly interpolate each second-level sharing $\widetilde{f}_1(x), \ldots, \widetilde{f}_n(x)$, but the interpolated values $\widetilde{f}_1(0), \ldots, \widetilde{f}_n(0)$ do not define a valid (first-level) Shamir-sharing of degree t,[24] then the following steps are executed (in order to help an honest P_V find the dealer whose original first-level 1D-sharings are incorrect or to find out which P_i shared some incorrect $s_i^{(m)}$ in Step 1.).
      3.2.1 For every dealer P_D, the random linear combination $f^{(*,D)}(x)$ of his (original first-level) 1D-sharings is defined as $f^{(*,D)}(x) = \sum_{m=1}^{\ell} r^{(m)} f^{(m,D)}(x) + f^{(\ell+1,D)}(x)$. Accordingly, for every dealer P_D, every player P_i with $\{P_i, P_D\} \notin \Delta$ and $\{P_i, P_V\} \notin \Delta$ sends to P_V his share $s_i^{(*,D)}$ on $f^{(*,D)}(x)$, i.e., $s_i^{(*,D)} = \sum_{m=1}^{\ell} r^{(m)} s_i^{(m,D)} + s_i^{(\ell+1,D)}$.
      3.2.2 P_V checks for every player P_i with $\{P_V, P_i\} \notin \Delta$ that $\sum_{P_D : \{P_i, P_D\} \notin \Delta} s_i^{(*,D)} = \widetilde{f}_i(0)$.[25] If the check fails for some

[24]Note that $\widetilde{f}_i(0) = f_i(0)$ for every i, i.e., $\widetilde{f}_i(0)$ is the linear combination of the values that P_i did indeed 1D-share as his shares $s_i^{(m)}$ in Step 1, thus if $\widetilde{f}_1(0), \ldots, \widetilde{f}_n(0)$ are not t-consistent, it means that either one of the original 1D-sharings is incorrect or some player shared incorrect values as his shares $s_i^{(m)}$ in Step 1 .

[25]Note that the Kudzu-shares $s_i^{(*,D)}$ with $\{P_i, P_D\} \in \Delta$ are 0 and do not contribute to

P_i, then P_V broadcasts i, and the protocol fails with $E = \{P_V, P_i\}$.

3.2.3 P_V broadcasts the index D of P_D such that the received shares $s_i^{(*,D)}$ (for every i with $\{P_i, P_D\} \notin \Delta$ and $\{P_i, P_V\} \notin \Delta$) do not define a correct 1D-sharing.

3.2.4 Every $P_i \in \mathcal{P}$ with $\{P_i, P_V\} \notin \Delta$ and $\{P_i, P_D\} \notin \Delta$ broadcasts his share $s_i^{(*,D)}$.

3.2.5 If the broadcasted shares define a correct 1D-sharing for dealer P_D, then P_V broadcasts the index i of the player P_i with $\{P_V, P_i\} \notin \Delta$ who has broadcast a different share $s_i^{(*,D)}$ than he has privately sent to P_V in Step 3.2.1, and the protocol fails with $E = \{P_V, P_i\}$. If the broadcasted shares do not define a correct 1D-sharing for dealer P_D, then P_D broadcasts the index i of a player P_i with $\{P_D, P_i\} \notin \Delta$ who broadcasted a wrong share $s_i^{(*,D)}$, and the protocol fails with $E = \{P_D, P_i\}$.

Lemma 17 *If* Upgrade1Dto2D *succeeds, then with overwhelming probability, the upgraded sharings are correct 2D-sharings. If the protocol fails, then the localized pair $E = \{P_i, P_j\}$ is new and contains at least one corrupted player. The privacy of the shared values is guaranteed through the whole protocol (even if it fails). The protocol communicates $\mathcal{O}(\ell n^2 + n^3)$ and broadcasts $\mathcal{O}(n)$ field elements.*

Proof: Along the lines of the proof of Lemma 14. ■

5.6.5 Information Checking with Dispute Control

An *information-checking (IC)* scheme allows a sender to deliver a message to a recipient in such a way that the recipient can later forward the message and prove its authenticity to a designated verifier. More precisely, an IC-scheme for a sender P_S, recipient P_R, and verifier P_V, consists of two protocols:[26]

the sum.

[26]In [RB89, CDD$^+$99], a different notation is used. They denote the sender as "dealer", the recipient as "intermediary", and the verifier as "receiver".

IC-Distr**:** The sender P_S delivers the message m and some authentication tag y to P_R and some checking tag z to P_V.

IC-Reveal**:** The recipient P_R forwards m and y to P_V, who uses z to verify the authenticity of m, and either accepts or rejects m.

Our information-checking protocol is a variant of the information-checking protocol of [CDD$^+$99] with two modifications. First, our IC-Distr protocol may fail in case of a fault; then, a dispute among two of the three players is identified.[27] Second, our protocol supports authenticating long messages $m = (m_1, \ldots, m_\ell) \in \mathbb{F}^\ell$ without additional costs.[28] We only assume that ℓ field elements $\zeta_1, \ldots, \zeta_\ell$ are fixed and publicly known.

For authenticating $m = (m_1, \ldots, m_\ell)$, a random degree-$\ell$ polynomial $f(x)$ with $f(\zeta_i) = m_i$ for $i = 1, \ldots, \ell$ is chosen, then the authentication tag is $y = f(0)$ and the verification tag is a random point $z = (u, v)$ with $u \neq \zeta_i$ $(\forall i)$ and $f(u) = v$. One can easily verify that this approach satisfies completeness, secrecy, and correctness (with error probability $\ell/(|\mathbb{F}| - \ell - 1)$) as long as the tags are computed as indicated. In order to ensure that the sender computes the tags correctly, we use a cut-and-choose proof: The sender generates and distributes κ independent tags, and the verifier hands half of them to the recipient, who checks them. The concrete protocols are given in the sequel:

Protocol IC-Distr**.**

1. PRIVATE COMPUTATION: The sender P_S, holding message $m = (m_1, \ldots, m_\ell)$, selects uniformly at random κ authentication tags $y_1, \ldots, y_\kappa \in_R \mathbb{F}^\kappa$, κ elements $u_1, \ldots, u_\kappa \in_R (\mathbb{F} \setminus \{0, \ldots, \ell\})^\kappa$, and computes $v_1, \ldots, v_\kappa$ such that for each $i \in \{1, \ldots, \kappa\}$, the $\ell + 2$ points $(0, y_i), (\zeta_1, m_1), \ldots, (\zeta_\ell, m_\ell), (u_i, v_i)$ lie on a polynomial of degree ℓ. P_S sends the message m and the authentication tags $y_1, \ldots, y_\kappa$ to P_R and the verification tags $z_1 = (u_1, v_1), \ldots, z_\kappa = (u_\kappa, v_\kappa)$ to P_V.
2. FAULT DETECTION:
   2.1 P_V partitions the index set $\{1, \ldots, \kappa\}$ into two partitions I and $\overline{I}$ of (almost) equal size, and sends I, $\overline{I}$, and z_i for every $i \in I$ to P_R.

[27]In our context, the IC-scheme will be used only by triples of players with no a priori dispute among them, so the identified dispute will be a new one.

[28]The costs in the scheme of [CDD$^+$99] grow linearly with the size of the message.

   2.2 P_R checks whether for *every* $i \in I$, the points $(0, y_i), (\zeta_1, m_1), \ldots, (\zeta_\ell, m_\ell), z_i$ lie on a polynomial of degree ℓ, and broadcasts either "accept" (and the protocol succeeded) or "reject".

3. FAULT LOCALIZATION: If P_R broadcasted "reject", the protocol fails and:

   3.1 P_R selects $i \in I$ such that the verification tag z_i received from P_V does not match with the message m and the authentication tag y_i received from P_S, and broadcasts i and z_i.

   3.2 P_S and P_V broadcast z_i.

   3.3 If the z_i-s broadcasted by P_S and P_V differ, then $E = \{P_S, P_V\}$. Otherwise, if the z_i-s broadcasted by P_R and P_V differ, then $E = \{P_R, P_V\}$. Otherwise, $E = \{P_S, P_R\}$.

Protocol IC-Reveal**.**

1. The recipient P_R sends the message m and the authentication tags y_i for $i \in \overline{I}$ to the verifier P_V.
2. The verifier with verification tags $z_1, \ldots, z_\ell$ accepts $m = (m_1, \ldots, m_\ell)$ if for *any* $i \in \overline{I}$, the points $(0, y_i), (\zeta_1, m_1), \ldots, (\zeta_\ell, m_\ell), z_i$ form a polynomial of degree ℓ; otherwise, he rejects m.

Lemma 18 *If* IC-Distr *succeeds and P_V, P_R are honest, then with overwhelming probability P_V accepts the message m in* IC-Reveal *(completeness). If* IC-Distr *fails, then the localized pair E contains at least one corrupted player. If P_S and P_V are honest, then with overwhelming probability, P_V rejects any fake message $m' \neq m$ in* IC-Reveal *(correctness). If P_S and P_R are honest, then P_V obtains no information about m in* IC-Distr *(even if it fails) (privacy).*

Proof: Completeness: If the cut-and-choose proof is successful, then the probability that at least one of the remaining authentication tags is valid is at least $1-\kappa/2^\kappa$. Correctness: The probability that an corrupted receiver can produce at least one correct tag for a message $m' \neq m$ is equal to the probability, that he can guess at least one verification point z_i, which is less than $\kappa/(2^\kappa - \ell - 1)$. Privacy follows from the fact that the verification tag is statistically independent from the message. ■

5.6.6 Upgrading 2D-Sharings to 2D*-Sharings

The following protocol upgrades ℓ 2D-sharings to 2D*-sharings. We denote the 2D-shared values by $s^{(m)}$ (for $m = 1, \ldots, \ell$), the shares of each player $P_i \in \mathcal{P}$ by $s_i^{(m)}$, and $P_j' s$ share-share of $s_i^{(m)}$ by $s_{ij}^{(m)}$.

Protocol Upgrade2Dto2D*.

1. For every triple of players $P_i, P_j, P_k \in \mathcal{P}$ with no dispute among them (i.e., $\{P_i, P_j\} \notin \Delta$, $\{P_i, P_k\} \notin \Delta$, $\{P_j, P_k\} \notin \Delta$), the protocol IC-Distr is invoked for the message $m = (s_{ij}^{(1)}, \ldots, s_{ij}^{(\ell)})$ with sender P_i, receiver P_j and verifier P_k. The message is not really sent, as P_j already holds it. Furthermore, these up to n^3 parallel invocations are merged when it comes to fault-detection and fault-localization: Every player P_j broadcasts one single bit in the fault-detection, indicating whether he observed a fault in one of the instances he acted as recipient. Then, the smallest player P_j that reported a fault, broadcasts i and k, indicating the instance i, j, k in which he observed the fault, and fault-localization is invoked only for this instance.

Lemma 19 *If the 2D-sharings to be upgraded are correct (for the actual Δ) and the protocol* Upgrade2Dto2D* *succeeds, then the upgraded 2D*-sharings are with overwhelming probability correct. If the protocol fails, then the output pair E is new and contains at least one corrupted player. The privacy of the shared values is guaranteed through the whole protocol (even if it fails). The protocol communicates $\mathcal{O}(n^3\kappa)$ and broadcasts $\mathcal{O}(n)$ field elements.*

5.6.7 ABC-Protocol

The following protocol allows every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ to prove that for every $m = 1, \ldots, \ell$, the (for the actual Δ correctly) 1D-shared value $c^{(m,k)}$ is the product of the (for the actual Δ correctly) 1D-shared values $a_k^{(m)}$ and $b_k^{(m)}$. This ABC-protocol is inspired by the corresponding protocol of [CDD+99].

The intuition of the ABC protocol is the following (where we denote the factors as a and b and the product as c): The prover shares a random $\overline{a}$ and $\overline{c} = \overline{a}b$, i.e., $(\overline{a}, b, \overline{c})$ is a multiplication triple, and proves for a random challenge r, that the shared triple $(ra + \overline{a}, b, rc + \overline{c})$ is a correct multiplication triple. This is achieved by first reconstructing $\widetilde{a} = ra + \overline{a}$, and then verifying that $z = \widetilde{a}b - rc - \overline{c}$ is a sharing of 0. For the sake of efficiency, we parallelize this ABC-proof for many triples and amortize the verification. Instead of reconstructing the sharing of each $\widetilde{a}$, we ask the prover to send the (alleged) values $\widetilde{a}$ to every player; who then verify that a random linear combination of these sharings reconstructs to the linear combination of the alleged values. Analogously, instead of verifying each z to be zero, the players reconstruct a random linear combination of these values, which must be zero.

Protocol ABC**.**

1. Every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ selects for each $m = 1, \dots, \ell$ a random $\overline{a}_k^{(m)}$ and computes $\overline{c}^{(m,k)} = \overline{a}_k^{(m)} b_k^{(m)}$.
2. Invoke VSS1D to let every $P_k \in \mathcal{P} \setminus \mathcal{X}$ verifiably 1D-share $\overline{a}_k^{(m)}$ and $\overline{c}^{(m,k)}$ for $m = 1, \dots, \ell$.
3. Invoke GenerateChallenges to generate one random challenge r.
4. Every $P_k \in \mathcal{P} \setminus \mathcal{X}$ sends $\widetilde{a}_k^{(m)} = ra_k^{(m)} + \overline{a}_k^{(m)}$ for $m = 1, \dots, \ell$ to every $P_i \in \mathcal{P}$ with $\{P_k, P_i\} \notin \Delta$.
5. Invoke GenerateChallenges to generate ℓ challenges $r^{(1)}, \dots, r^{(\ell)}$.
6. Invoke Reconstruct1D with $\mathcal{P}_R = \mathcal{P} \setminus \mathcal{X}$ to publicly reconstruct $\widehat{a}_k = \sum_{m=1}^{\ell} r^{(m)} \left(ra_k^{(m)} + \overline{a}_k^{(m)} \right)$ for $k = 1, \dots, n$.[29]
7. Every $P_i \in \mathcal{P} \setminus \mathcal{X}$ checks for every P_k with $\{P_i, P_k\} \notin \Delta$ whether $\widehat{a}_k = \sum_{m=1}^{\ell} r^{(m)} \widetilde{a}_k^{(m)}$, and broadcasts the index k of a player P_k for whom the check failed, respectively $\perp$ if all checks succeed. If at least one player P_i broadcasts k with $\{P_i, P_k\} \notin \Delta$, then the protocol fails with $E = \{P_i, P_k\}$ for the smallest such P_i (and the accused P_k).
8. Invoke Reconstruct1D with $\mathcal{P}_R = \mathcal{P} \setminus \mathcal{X}$ to reconstruct $z^{(k)} = \sum_{m=1}^{\ell} r^{(m)} \left(\widetilde{a}_k^{(m)} b_k^{(m)} - rc^{(m,k)} - \overline{c}^{(m,k)} \right)$ for $k = 1, \dots, n$. Note that $\widetilde{a}_k^{(m)}$ is a constant known to all players P_i with $\{P_i, P_k\} \notin \Delta$,[30] hence $z^{(k)}$ is a linear combination of 1D-shared values, as required by Reconstruct1D. Note that when this reconstruction succeeds, then every player $P_V \in \mathcal{P} \setminus \mathcal{X}$ reconstructs the same vector $(z^{(1)}, \dots, z^{(n)})$.

[29]Note that the 1D-sharing $\widehat{a}_k$ belongs to dealer P_k. Formally, Reconstruct1D requires every value to be reconstructed to be the *sum* of one 1D-sharing of each dealer in $\mathcal{P}_D$; hence, we implicitly assume constant-0 1D-sharings for the other dealers, and set $\mathcal{P}_D = \mathcal{P} \setminus \mathcal{X}$.

[30]Note that P_k is the owner of the 1D-sharing of $z^{(k)}$; hence, the share of every player P_i with $\{P_i, P_k\} \in \Delta$ is Kudzu, and he does not need to know the constant $\widetilde{a}_k^{(m)}$.

9. Every player $P_V \in \mathcal{P} \setminus \mathcal{X}$ checks whether the reconstructed values $z^{(k)} = 0$ for every $P_k \in \mathcal{P} \setminus \mathcal{X}$. If this check fails, then P_k is corrupted, and the protocol fails with $E = \{P_i, P_k\}$ for all $P_i \in \mathcal{P}$ (i.e., P_k is in dispute with every player).

Lemma 20 *If all triples $(a_k^{(m)}, b_k^{(m)}, c^{(m,k)})$ are correctly 1D-shared for the actual Δ, then the following holds with overwhelming probability: If* ABC *succeeds, then the checked triples $(a_k^{(m)}, b_k^{(m)}, c^{(m,k)})$ are correct multiplication triples, i.e. $c^{(m,k)} = a_k^{(m)} b_k^{(m)}$ for every $m = 1, \ldots \ell$, and their privacy is preserved. If the protocol fails, then it localizes a new dispute pair E containing at least one corrupted player (respectively localizes single player who is corrupted). The protocol communicates $\mathcal{O}(\ell n^2 + n^3)$ and broadcasts $\mathcal{O}(n)$ field elements.*

Proof: In order to prove correctness, assume that there is at least one (incorrect) triple $(a_k^{(m)}, b_k^{(m)}, c^{(m,k)})$ (of player P_k) such that $c^{(m,k)} \neq a_k^{(m)} b_k^{(m)}$. Then there is at most one (out of 2^κ) challenge $r \in F$ such that $(ra_k^{(m)} + \overline{a}_k^{(m)})b_k^{(m)} - rc^{(m,k)} - \overline{c}^{(m,k)} = 0$. If $(ra_k^{(m)} + \overline{a}_k^{(m)})b_k^{(m)} - rc^{(m,k)} - \overline{c}^{(m,k)} \neq 0$ then there are at most $2^{\kappa(\ell-1)}$ (out of $2^{\kappa\ell}$) challenge vectors $(r^{(1)}, \ldots, r^{(\ell)}) \in \mathbb{F}^\ell$ such that the sum $z^{(k)} = \sum_{m=1}^{\ell} r^{(m)} \left(\left(ra_k^{(m)} + \overline{a}_k^{(m)} \right) b_k^{(m)} - rc^{(m,k)} - \overline{c}^{(m,k)} \right) = 0$. So provided that the values $a_k^{(m)}, b_k^{(m)}, c^{(m,k)}, \overline{a}_k^{(m)}, \overline{c}^{(m,k)}$ for $m = 1, \ldots, \ell$ are correctly 1D-shared, the challenges are random, and in Step 4., player P_k sent the correct $\widetilde{a}_k^{(m)} = ra_k^{(m)} + \overline{a}_k^{(m)}$ for $m = 1, \ldots, \ell$ to every $P_i \in \mathcal{P}$ with $\{P_k, P_i\} \notin \Delta$, the probability of the false triple not being detected is at most $2/2^\kappa$, which is negligible. As with overwhelming probability the values $a_k^{(m)}, b_k^{(m)}, c^{(m,k)}, \overline{a}_k^{(m)}, \overline{c}^{(m,k)}$ for $m = 1, \ldots, \ell$ are correctly 1D-shared and the challenges are random, it is now sufficient to show that the probability of P_k sending at least one false $\widetilde{a}_k^{(m)} \neq ra_k^{(m)} + \overline{a}_k^{(m)}$ to at least one honest verifier P_i in Step 4 and not being detected (by P_i) in Step 7 is negligible. This holds because for a false $\widetilde{a}_k^{(m)}$ there are at most $2^{\kappa(\ell-1)}$ (out of $2^{\kappa\ell}$) challenge vectors for which the check in Step 7 does not fail. ■

5.7 Preparation Phase

The goal of this phase is to generate c_M random 2D*-shared multiplication triples (a, b, c) (one for each multiplication gate) and c_R random 2D*-shared values (one for each random gate). We wastefully generate $c_M + c_R$ random multiplication triples and use only the first factor for the random gates.

The generation of the $c_M + c_R$ multiplication triples is divided into n^2 segments, each of length $L = \lceil (c_M + c_R)/n^2 \rceil$. The computation is non-robust, and its correctness is verified at the end of the segment. In fact, the segment will consist of several stages, each with a private computation and fault-detection. As soon as a fault is reported in a fault-detection procedure, the corresponding fault-localization is used to localize a new dispute to be registered in Δ, and the whole segment has failed and is repeated.

Protocol PreparationPhase**.**

Set $\Delta := \{\}$ and $\mathcal{X} = \{\}$, and for each segment (of length L) do the following steps. If any of the invoked sub-protocols fails, then include the localized pair $E = \{P_i, P_j\}$ in Δ, i.e., $\Delta \leftarrow \Delta \cup \{P_i, P_j\}$, and repeat the failed segment.

1. Generate $2L$ correct random 2D-sharings $\left(a^{(1)}, b^{(1)}\right), \ldots, \left(a^{(L)}, b^{(L)}\right)$:

   1.1. Every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ 1D-shares L randomly selected pairs $\left(a^{(1,k)}, b^{(1,k)}\right), \ldots, \left(a^{(L,k)}, b^{(L,k)}\right) \in \mathbb{F}^2$ among the players. We denote the distributed shares of $a^{(m,k)}$ by $a_1^{(m,k)}, \ldots, a_n^{(m,k)}$.

   1.2. Invoke Upgrade1Dto2D with $\mathcal{P}_D = \mathcal{P} \setminus \mathcal{X}$ and $\ell = L$ to upgrade the implicitly defined sum sharings of $\sum_{P_k \in \mathcal{P}_D} a^{(1,k)}, \ldots, \sum_{P_k \in \mathcal{P}_D} a^{(L,k)}$ to 2D-sharings, resulting in L correctly 2D-shared random values $a^{(1)}, \ldots, a^{(\ell)}$. The same for b.

2. Multiply the L pairs $\left(a^{(1)}, b^{(1)}\right), \ldots, \left(a^{(L)}, b^{(L)}\right)$, resulting in L correctly 2D-shared products $c^{(1)}, \ldots, c^{(L)}$:

   2.1. Every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ computes for every $m = 1, \ldots, L$ the product $c^{(m,k)}$ of his shares $a_k^{(m)}$ and $b_k^{(m)}$. Note that the product $c^{(m)} = a^{(m)} b^{(m)}$ can be computed as a weighted sum of these values $c^{(m,k)}$ (namely Lagrange interpolation); accordingly, we will compute a sharing of $c^{(m)}$ as weighted sum of sharings of $c^{(m,1)}, \dots, c^{(m,n)}$.

2.2. Invoke VSS1D to let every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ verifiably 1D-share his values $c^{(1,k)}, \dots, c^{(L,k)}$.

2.3. Invoke the protocol ABC to have every player $P_k \in \mathcal{P} \setminus \mathcal{X}$ prove that for every $m = 1, \dots, L$, the value $c^{(m,k)}$ he shared in Step 2 is indeed the product of his shares $a_k^{(m)}$ and $b_k^{(m)}$, which are implicitly 1D-shared as part of the 2D-sharings of $a^{(m)}$ and $b^{(m)}$, respectively.

2.4. Invoke the protocol Upgrade1Dto2D with $\mathcal{P}_D = \mathcal{P} \setminus \mathcal{X}$ to upgrade the sharings of the weighted sums $\sum_{P_k \in \mathcal{P}_D} \lambda_k c^{(1,k)}, \dots, \sum_{k=1}^{n} \lambda_k c^{(L,k)}$ to 2D-sharings, where λ_k denotes the Lagrange coefficients.[31]

3. Invoke Upgrade2Dto2D* to upgrade all $3L$ 2D-sharings to 2D*-sharings.

Lemma 21 *With overwhelming probability, the protocol* PreparationPhase *generates $c_M + c_R$ correctly 2D*-shared random multiplication triples (a, b, c) with $c = ab$; the secrecy of the triples is preserved. The protocol communicates $\mathcal{O}((c_M + c_R)n^2 + n^5\kappa)$ and broadcasts $\mathcal{O}(n^3)$ field elements.*

Proof: In order to show the correctness first consider one execution of the Steps 1.–3. for one segment of length L. (Note that the dispute set Δ remains unchanged through Steps 1.–3.) If the execution succeeds, then with overwhelming probability, the triples $\left(a^{(1)}, b^{(1)}, c^{(1)}\right), \dots, \left(a^{(L)}, b^{(L)}, c^{(L)}\right)$ are correctly 2D*-shared (because of Lemma 14, 17, and 19), and $c = ab$ holds because of Lemma 20 for each triple (a, b, c). As there are n^2 segments and the adversary can provoke less than n^2 executions to fail (in total), he has less then $2n^2$ attempts to introduce a segment with a false triple. Because n is at most polynomial in κ, the probability that a false triple is not detected is negligible.

Privacy follows from the privacy of the invoked sub-protocols. Some of them do not guarantee privacy in case of a failure, but in such case all generated values are discarded and completely new shared values will be generated. ∎

[31]Note that the sharings of detected players $P_D \in \mathcal{X}$ are not considered in the Lagrange interpolation; however, as their shares are 0 (Kudzu), this omission does not falsify the outcome.

5.8 Input Phase

The goal of the input phase is to provide 2D*-sharings of c_I inputs.

We set the upper bound on the number of input gates of a segment to $L = \lceil \frac{c_I}{n^2} \rceil$ and limit each segment to contain only input gates of the same player.

Protocol InputPhase**.**

For each segment, the following steps are executed to let the dealer $P_D \in \mathcal{P} \setminus \mathcal{X}$ verifiably 2D*-share his L inputs $s^{(1)}, \ldots, s^{(L)}$.[32] If any of the invoked sub-protocols fails, include the localized pair $E = \{P_i, P_j\}$ in Δ, i.e., $\Delta \leftarrow \Delta \cup \{P_i, P_j\}$, and repeat the segment.

1. P_D (unverifiably) 1D-shares the input values $s^{(1)}, \ldots, s^{(L)}$.
2. Invoke Upgrade1Dto2D with $\mathcal{P} = \{P_D\}$ to upgrade the 1D-sharings of $s^{(1)}, \ldots, s^{(L)}$ to 2D-sharings.
3. Invoke Upgrade2Dto2D* to upgrade the 2D-sharings of $s^{(1)}, \ldots, s^{(L)}$ to 2D*-sharings.

Lemma 22 *With overwhelming probability, the protocol* InputPhase *computes correct 2D*-sharings of c_I inputs, where the privacy of the inputs of the honest players is preserved. The protocol communicates $\mathcal{O}(c_I n^2 + n^5 \kappa)$ and broadcasts $\mathcal{O}(n^3)$ field elements.*

Proof: In one execution of Steps 1.–3., the probability of success in spite of a false sharing is negligible. As there are at most $n^2 + n$ segments and less than n^2 repetitions, the adversary has at most $2n^2 + n$ independent attempts to introduce a segment with a false sharing, hence his success probability is negligible. The privacy is guaranteed even in case of failure (and repetition) of some segment. ■

[32]If the dealer P_D is detected, i.e., $P_D \in \mathcal{X}$, then the players take the all-zero sharing of 0, i.e., every share is 0 and every share-share is 0 (Kudzu). Note that no authentication tags are needed because all share-shares are Kudzu.

5.9 Computation Phase

The computation of the circuit proceeds gate-by-gate. First, to every random and every multiplication gate, a prepared 2D*-shared random triple is assigned.

Given the 2D*-sharings of the multiplication triples and of the inputs, all values to be computed (and to be opened) in the computation stage are completely determined. We therefore call the values shared in the preparation phase and in the input phase the *base values* of the computation. All base values are robustly shared with 2D*-sharings.

It turns out that the value of each gate can be computed as linear combination of such base values. This is trivial as long as the circuit only consists of addition and random gates. For a multiplication gate, the players publicly reconstruct two sharings (both linear combinations of base values), such that the value of the multiplication gate is a linear combination of base values, where the coefficients of the linear combination depend on the two reconstructed values [Bea91a]. Hence, the whole computation phase consists only of a sequence of reconstructions of publicly known linear combinations of base sharings. More precisely, the gates are evaluated as follows:

Input Gate: Assign the corresponding 2D*-sharing of the input to the gate.

Random Gate: Assign the 2D*-sharing of a of the assigned multiplication triple (a, b, c) to the gate.

Addition Gate: To both summands, a linear combination of base sharings was assigned. Assign to the gate the sum of these two linear combinations (which is again a linear combination of base sharings).

Multiplication Gate: To both factors, a linear combination of base sharings was assigned. We denote the corresponding values by x and y, and denote the assigned multiplication triple by (a, b, c). The players reconstruct $d_x = x - a$ and $d_y = y - b$ towards every player in $\mathcal{P}$ (both d_x and d_y are represented as known linear combination of base sharings), and assign to the gate the linear combination $d_x d_y + d_x b + d_y a + c$ (i.e., a linear combination of the 2D*-sharings of a, b, and c, all three of them base sharings).

Output Gate: The players reconstruct the assigned linear combination of base sharings towards the designated output player.

Now, we are left with the problem of opening known linear combinations of base values towards designated players. For every multiplication gate, we need $2n$ reconstructions (one towards every player), and for every output gate, we need 1 reconstruction. Hence, in total we need to reconstruct $2nc_M + c_O$ linear combinations of 2D*-sharings. This job is, as usual, divided into n^2 segments, each with at most $L = \lceil (2nc_M + c_O)/n^2 \rceil$ reconstructions. Each reconstruction is processed non-robustly, and at the end of the segment, the players verify that no fault has occurred. In the non-robust reconstruction the receiver either obtains the right value, or he observes a fault, stops the further processing of this segment and only joins again in the fault handling procedure.

Protocol ComputationPhase**.**

For each segment with L reconstructions, the following steps are executed. If in a segment a fault is detected in Step 2., then Step 3 is executed to localize a new dispute pair E, which is included in Δ, i.e., $\Delta \leftarrow \Delta \cup \{E\}$, and the failed segment is repeated.

1. PRIVATE COMPUTATION: Execute the following for each output operation.[33] Denote the designated output player with P_k, the publicly known linear combination for the output operation with $\mathcal{L}$, and the 2D*-shared base values used in the linear combination with $s^{(1)}, s^{(2)}, \ldots$. Furthermore, we denote the share and shares-shares of P_i by $s_i^{(m)}, s_{1i}^{(m)}, \ldots, s_{ni}^{(m)}$, respectively, and the polynomial used for the second-level sharing of $s_i^{(m)}$ by $f_i^{(m)}(x)$.
   1.1 Every P_i with $\{P_i, P_k\} \notin \Delta$ sends his linearly combined share $s_i = \mathcal{L}(s_i^{(1)}, s_i^{(2)}, \ldots)$ to P_k, who receives a message in $\mathbb{F} \cup \{\epsilon\}$.[34]
   1.2 If P_k received *all* shares s_i he was supposed to get (i.e., there was no empty message ϵ), and the received shares lie on a polynomial $f(x)$ of degree t, he computes the output value as $s = f(0)$; otherwise P_k observes a fault and aborts the segment, i.e., for the rest of the segment, P_k only sends empty messages.

[33]All output operations at the same level in the circuit can be executed in parallel.

[34]It is legal for an honest player P_i to send the empty message ϵ to P_k, namely when P_i has observed a fault in an earlier gate. Hence, P_k must accept the empty message as valid.

2. FAULT DETECTION: Every player $P_i \in \mathcal{P} \setminus \mathcal{X}$ broadcasts the index q_i of the first failed reconstruction operation, respectively $\perp$ if he successfully completed the segment. If all players broadcast $\perp$, then the evaluation of the current segment succeeded
3. FAULT LOCALIZATION: Execute the following steps for the player P_k with the smallest q_k, for the failed reconstruction operation with index q_k:
   - 3.1 Every player P_i with $\{P_k, P_i\} \notin \Delta$ sends the polynomial $f_i(x) = \mathcal{L}(f_i^{(1)}(x), f_i^{(2)}(x), \ldots)$ and all share-shares $s_{ji}(x) = \mathcal{L}(s_{ji}^{(1)}(x), s_{ji}^{(2)}(x), \ldots)$ to P_k.
   - 3.2 If for some P_i with $\{P_k, P_i\} \notin \Delta$, P_k did not receive s_i in Step 1.1, or the provided polynomial $f_i(x)$ is inconsistent with s_i (i.e., $f_i(0) \neq s_i$), then P_k broadcasts i, and the fault localization terminates with $E = \{P_k, P_i\}$.
   - 3.3 P_k identifies two players P_i, P_j with $\{P_k, P_i\} \notin \Delta$ and $\{P_k, P_j\} \notin \Delta$, such that $f_i(\alpha_j) \neq s_{ij}$,[35] and broadcasts $(i, j, s_{ij}, f_i(\alpha_j))$.
   - 3.4 Both P_i and P_j broadcast a bit indicating whether or not they agree with the values broadcasted by P_k. If P_i (respectively P_j) disagrees, the fault localization terminates with $E = \{P_k, P_i\}$ (respectively $E = \{P_k, P_j\}$).
   - 3.5 As both P_i and P_j agree with s_{ij} respectively $f_i(\alpha_j)$ as broadcasted by P_k, and as $f_i(\alpha_j) \neq s_{ij}$, either P_i or P_j delivered a wrong value to P_k. P_j can use the information checking scheme to prove to P_k the correctness of s_{ij}. However, there are no authentication tags for s_{ij} itself, but s_{ij} is computed as a publicly known linear combination $\mathcal{L}$ of base sharings, for which authentication tags exist (one authentication tag for all share-shares x_{ij} of each segment), respectively which are Kudzu and hence publicly known. Hence, P_j executes the protocol IC-Reveal for revealing the provably correct share-shares x_{ij} of every base sharing x, and if P_k accepts all invocations and the linear combination on the share-shares yields s_{ij}, then P_k broadcasts i and $E = \{P_k, P_i\}$, otherwise, P_k broadcasts j and $E = \{P_k, P_j\}$.

Lemma 23 *If all base values are correctly 2D*-shared and all multiplication triples are correct and random, then with overwhelming probability, the circuit*

[35] The existence of such a pair (P_i, P_j) is guaranteed due to the correctness of the base 2D*-sharings.

evaluation as described above is correct, robust and private. The protocol communicates $\mathcal{O}((c_I n^2 + c_M n^2 + c_R n^2 + c_O n + n^4)\kappa)$ and broadcasts $\mathcal{O}(n^3)$ field elements.

Proof: Once the base values are correctly 2D*-shared, the computation phase is purely deterministic. An honest player will never reconstruct a wrong secret: He receives shares from all players he is not in dispute with (otherwise he does not reconstruct at all), hence there are at least $t + 1$ correct shares from the honest players which prevent him from reconstructing a wrong value. Hence, the adversary cannot falsify the outputs of honest players, he can only prevent them from reconstructing. In this case, a fault is detected, a new dispute is localized and included in Δ, and the segment is repeat till eventually all honest players reconstruct all their outputs.

In order to argue about the privacy of the protocol, we observe that share-shares x_{ij} are revealed only when P_i and P_j disagree on some value s_{ij}, hence either P_i or P_j is corrupted. By revealing these values, the adversary obtains no additional information. ■

5.10 Main Protocol

The MPC protocol consists of the three described phases:

Protocol MPC**.**

1. Invoke PreparationPhase to prepare $c_M + c_R$ random 2D*-shared multiplication triples.
2. Invoke InputPhase to provide 2D*-sharings of the c_I inputs.
3. Invoke ComputationPhase to compute and reconstruct the outputs towards the specified players.

Theorem 5 *A set of n players communicating over a secure synchronous network, can evaluate an agreed function of their inputs securely against an unbounded active adaptive adversary corrupting up to $t < n/2$ of the players with communicating $\mathcal{O}(c_I n^2 + c_M n^2 + c_R n^2 + c_O n + n^5\kappa)$ field elements and broadcasting $\mathcal{O}(n^3)$ field elements, where c_I, c_M, c_R, c_O denote the number of input gates, multiplication gates, random gates, and output gates, respectively.*

Note that for large enough circuits, the costs for simulating the $\mathcal{O}(n^3)$ broadcast invocations are dominated by the normal communication costs, such that the overall communication complexity is (up to a constant factor) the same as the one of passively secure MPC protocols [BGW88].

However, for very small circuits, the $\mathcal{O}(n^3)$ broadcasts are dominating the overall costs. Note that even in this case, our protocol is substantially more efficient than the most efficient previously known protocol for the same model [CDD$^+$99], which broadcasts $\Omega(n^5)$ field elements *per multiplication*.

Chapter 6

Byzantine Agreement with Faulty Minority

6.1 Introduction

Byzantine Agreement (BA) among n players allows the players to agree on a value, even when up to t of the players are faulty.

In the broadcast variant of BA, one dedicated player holds a value, and all players shall learn this value. In the consensus variant of BA, every player holds (presumably the same) value, and the players shall agree on this value.

BA is an important primitive widely used in distributed protocols, hence its efficiency is of particular importance.

BA from scratch, i.e., without a trusted setup, is possible only for $t < n/3$. In this setting, the known BA protocols are highly efficient ($\mathcal{O}(n^2)$ bits of communication) and provide information-theoretic security.

When a trusted setup is available, then BA is possible for $t < n/2$ (consensus), respectively for $t < n$ (broadcast). In this setting, only computationally secure BA protocols are reasonably efficient ($\mathcal{O}(n^3\kappa)$ bits). When information-theoretic security is required, the most efficient known BA protocols require $\mathcal{O}(n^{17}\kappa)$ bits of communication per BA, where κ denotes a security parameter. The main reason for this huge communication is that in the information-theoretic world, parts of the setup are *consumed* with every invocation to BA, and hence the setup must be refreshed. This refresh operation is highly complex and communication-intensive.

In this chapter we present BA protocols (both broadcast and consensus) with information-theoretic security for $t < n/2$, communicating $\mathcal{O}(n^5\kappa)$ bits per BA.

Our BA protocols are based on the broadcast protocol of [DS83], ameliorated with information-theoretically secure signatures [SHZI02].

6.2 Model

We consider a set of n players $\mathcal{P} = \{P_1, \ldots, P_n\}$, communicating over pairwise secure synchronous channels. Our protocols work in a finite field $\mathbb{F} = \mathrm{GF}(2^\kappa)$ where κ is a security parameter (we allow a negligible error probability of $\mathcal{O}(2^{-\kappa})$). To every player $P_i \in \mathcal{P}$, a unique non-zero element $\alpha_i \in \mathbb{F} \setminus \{0\}$ is assigned.

The faultiness of players is modeled by a central computationally-unlimited adversary who adaptively actively corrupts up to $t < n/2$ of the players.

We assume that there is a trusted setup, i.e., in an initialization phase, a fixed probabilistic function $\mathsf{Init} : 1^\kappa \rightarrow (\mathsf{state}_1, \ldots, \mathsf{state}_n)$ is run, and every player $P_i \in \mathcal{P}$ secretly receives state_i as his initial state.

6.3 Information-Theoretically Secure Signatures

A classical (cryptographic) signature scheme consists of three algorithms: KeyGen, Sign, and Verify. KeyGen generates two keys, a *signing key* for the signer and a public *verification key*; Sign computes a signature for a given message and a given signing key; and Verify checks whether a signature matches a message for a given verification key. A secure signature scheme must satisfy that every signature created by Sign is accepted by Verify (with the corresponding signing/verification keys, completeness), and without the signing key it is infeasible to compute a signature which is accepted by Verify (unforgeability). Classical signature schemes provide cryptographic security only, i.e., an unbounded forger can always find an accepting signature for any given message, with exhaustive search, using Verify as test predicate.

As an information-theoretically secure signature scheme must be secure even with respect to a computationally unbounded adversary, every verifier must have a different verification key (and these verification keys must be kept private). Hence there exist signatures that are valid for one verifier while being invalid for another verifier. However, such signatures should be (almost) impossible to find. Therefore, an additional property called *transferability* is required: It is impossible (except with a negligible probability) for a faulty signer to produce a signature which is valid for some honest verifier without being valid for some other honest verifier. We say that a signature scheme is information-theoretically secure if it is complete, unforgeable and transferable.

In [SHZI02], a so called (ψ, ψ')-secure signature scheme is presented, which allows the signer to sign a message $m \in \mathbb{F}$ such that any of the players in $\mathcal{P}$ can verify the validity of the signature. As long as the signer signs at most ψ messages and each verifier verifies at most ψ' signatures the success probability of attacks is less then $1/|\mathbb{F}| = 2^{-\kappa}$.

Our BA protocols use a one-time signature scheme (i.e., one setup allows only for one single signature), where every verifier may verify up to $t + 2$ signatures (of the same signer). In the context of [SHZI02], this means that we set $\psi = 1$ and $\psi' = t + 2$. By simplifying the notation (and by assuming that $2t + 1 \leq n$), we obtain the following scheme:

KeyGen: Key generation takes as input the string 1^κ, and outputs the signing key sk to the signer P_S and the n verification keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$ to the respective verifiers $P_1, \ldots, P_n$. The signing key is a random vector $\mathsf{sk} = (p_0, \ldots, p_{n+1}, q_0, \ldots, q_{n+1}) \in \mathbb{F}^{2(n+2)}$, defining the polynomial

$$\begin{aligned} F_{\mathsf{sk}}(V_1, \ldots, V_{n+1}, M) &= \left(p_0 + \sum_{j=1}^{n+1} p_j V_j\right) + M \left(q_0 + \sum_{j=1}^{n+1} q_j V_j\right) \\ &= p_0 + M q_0 + \sum_{j=1}^{n+1} (p_j + M q_j)\, V_j. \end{aligned}$$

The verification key vk_i of each player $P_i \in \mathcal{P}$ is the vector $\mathsf{vk}_i = (v_{i,1}, \ldots, v_{i,n+1}, x_i, y_i)$, where the values $v_{i,1}, \ldots, v_{i,n+1}$ are chosen uniformly at random from $\mathbb{F}$, and the x_i- and y_i-values characterize the polynomial F_{sk}, when applied to $v_{i,1}, \ldots, v_{i,n+1}$, i.e., $x_i = p_0 + \sum_{j=1}^{n+1} p_j v_{i,j}$ and $y_i = q_0 + \sum_{j=1}^{n+1} q_j v_{i,j}$.

Sign**:** The signature σ of a message $m \in \mathbb{F}$ is a vector $\sigma = (\sigma_0, \ldots, \sigma_{n+1})$, characterizing the polynomial F_{sk} when applied to m, i.e., $\sigma_j = p_j + mq_j$ for $j = 0, \ldots, n+1$.

Verify**:** Given a message m, a signature $\sigma = (\sigma_0, \ldots, \sigma_{n+1})$, and the verification key $\mathsf{vk}_i = (v_{i,1}, \ldots, v_{i,n+1}, x_i, y_i)$ of player P_i, the verification algorithm checks whether

$$x_i + my_i \overset{?}{=} \sigma_0 + \sum_{j=1}^{n+1} \sigma_j v_{i,j} \qquad \Big(= F_{\mathsf{sk}}(v_{i,1}, \ldots, v_{i,n+1}, m) \Big) .$$

The protocol has the following sizes: Signing key: $(2n + 4)\kappa$ bits; verification key: $(n + 3)\kappa$ bits; signature: $(n + 2)\kappa$ bits. The total information distributed for one signature scheme (called *sig-setup*) consists of $(n^2 + 5n + 8)\kappa$ bits.

Note that a sig-setup for the player set $\mathcal{P}$ is trivially also a valid sig-setup for every player subset $\mathcal{P}' \subseteq \mathcal{P}$. We will need this observation later.

6.4 Protocol Overview

Basically, the new broadcast protocol is the protocol of [DS83], ameliorated with information-theoretically secure signatures [SHZI02]. Similarly to [PW96], we start with a compact (constant-size) setup, which allows only for few broadcasts, and use some of these broadcasts for broadcasting the payload, and some of them to refresh the remaining setup, resulting in a fresh, full-fledged setup.

We use the player-elimination framework [HMP00] to substantially speed-up the refresh protocol: The generation of the new setup is performed non-robustly, i.e., it may produce a faulty setup when an adversary is present, this will however be detected by at least one honest player (who gets unhappy). At the end of the refresh protocol, the players jointly decide (using one BA-operation) whether the refresh has succeeded or not; if yes, they are happy to have generated a new setup. If it failed, they run a fault-handling procedure, which yields a set E of two players, (at least) one of them faulty. As originally the set $\mathcal{P}$ contains an honest majority, also the set $\mathcal{P} \setminus E$ contains an honest majority. So the player set is reduced to $\mathcal{P}' \leftarrow \mathcal{P} \setminus E$ (with at most $t' \leftarrow t - 1$ faulty players).

We are still missing the fresh setup; however, as with each fault-handling, one faulty player is eliminated from the actual player set, faults can occur only t times. For these t cases, we have a stock of t prepared setups, and with each fault, we take one out of this stock. This way it is ensured that at any point in the protocol, we have t' prepared setups on stock, where t' is the maximum number of faulty players in $\mathcal{P}'$. More precisely, the protocol runs as follows:

Initial Setup: The initial setup consists of $2 + 5t$ BA-setups[36]one for the first BA operation, one for the first invocation of the refresh protocol, and t extra setups for the stock, each consisting of 2 BA-setups for replacing the failed refresh and 3 BA-setups for localizing the set $E \subseteq \mathcal{P}$ in the fault-handling procedure. The actual player set is set to $\mathcal{P}' = \mathcal{P}$ and the maximum number of faulty players in $\mathcal{P}'$ to $t' = t$.

Broadcast/Consensus: To perform a BA operation, the protocol Broadcast, resp. Consensus is invoked with the payload. In parallel, Refresh is invoked to refresh the reduced setup. If successful, Refresh produces two BA-setups using only one single BA operation. If Refresh fails, 5 BA-setups are taken from the stock, an elimination set $E \subseteq \mathcal{P}'$ is localized (using 3 BA's) and eliminated ($\mathcal{P}' \leftarrow \mathcal{P}' \setminus E$, $t' \leftarrow t' - 1$), and the two remaining BA-setups are kept as new state – for the next Broadcast/Consensus operation.

6.5 Broadcast and Consensus

We present the protocols for the actual broadcast and consensus operation.

Note that the Refresh protocol outputs a correct BA setup for $\mathcal{P}'$ only (rather than $\mathcal{P}$). However, as $\mathcal{P} \setminus \mathcal{P}'$ might contain honest players we need to achieve BA in $\mathcal{P}$. We first present the BA protocols for $\mathcal{P}'$, then show how to realize BA in $\mathcal{P}$ using these protocols.

[36]BA-setup denotes the set of sig-setups used for one BA operation. As will become clear later, one BA-setup is equivalent to $2n$ sig-setups.

As [SHZI02] signatures can cope only with message in the field $\mathbb{F}$, also our BA protocols are limited to messages $m \in \mathbb{F}$. An extension to longer messages is sketched in Section 6.7.

We first present a broadcast protocol that allows a sender $P_S \in \mathcal{P}'$ to consistently distribute a message $m \in \mathbb{F}$ to the players in $\mathcal{P}'$.[37] The protocol is essentially the protocol of [DS83], with a simplified description of [Fit03]. In addition, the protocol is modified such that in one protocol run every player verifies at most $\psi' = t + 2$ signatures of each signer (as required by our signature scheme).

Every player maintains a set $\mathcal{A}$ of accepted messages, a set $\mathcal{N}$ of newly accepted messages, and (one or several) sets Σ_m of received signatures for a message m.

Protocol Broadcast**′.**

0. Sender P_S: Send m and the corresponding signature σ_S to all $P_i \in \mathcal{P}'$.
1. $\forall P_i \in \mathcal{P}'$: If P_i received from the sender a message m together with a valid signature σ_S set $\mathcal{A} = \mathcal{N} = \{m\}$ and $\Sigma_m = \{\sigma_S\}$. Otherwise set $\mathcal{A} = \mathcal{N} = \{\}$.

k. In each Step $k = 2, \ldots, t' + 1$, execute the following sub-steps for every player $P_i \in \mathcal{P}' \setminus \{P_S\}$:

   $k.1$ For every message $m \in \mathcal{N}$, compute the signature σ_i on m, and send $(m, \Sigma_m \cup \{\sigma_i\})$ to all players in $\mathcal{P}'$. Set $\mathcal{N} = \{\}$.

   $k.2$ In turn, for every message (m, Σ_m) received in Sub-step $k.1$ do:

   - If $m \in \mathcal{A}$, or if $|\mathcal{A}| \geq 2$, ignore the message,
   - else if Σ_m contains valid signatures from at least k different players in $\mathcal{P}'$, including P_S, include m in $\mathcal{A}$ and in $\mathcal{N}$,
   - else ignore the received message (m, Σ_m) and all further messages from the player who has sent it.

$t'+2$. $\forall P_i$: if $|\mathcal{A}| = 1$, then accept $m \in \mathcal{A}$ as the broadcasted value. Otherwise, the sender is faulty, and accept $m = \perp$ (or any fixed pre-agreed value from $\mathbb{F}$) as the broadcasted value.

[37]Note that Broadcast′ will not be used, it is presented only for the sake of clarity of the protocol Consensus′.

One can easily verify that the protocol Broadcast′ is as secure as the used signature scheme [DS83, Fit03] and that every player verifies at most $t+2$ signatures from the same signer. Furthermore, every signer P_i issues up to two signatures; however, the second one is for the sole goal of proving to other players that the sender P_S is faulty, and the secrecy of P_i's signing key is not required anymore. Hence, it is sufficient to use a one-time signature scheme, whose unforgeability property is broken once the signer issues two signatures.

To construct a consensus protocol in $\mathcal{P}'$, we use a trick of [Fit04]: Every player needs two sig-setups, a primary scheme for the same purpose as in the above protocol, and an alternative scheme for identifying the message (if there is any) originally held by all honest players. During the protocol execution, every player P_i additionally maintains (one or several) sets Σ'_m, containing alternative signatures σ'_j (issued by P_j) for m, where Σ'_m with $|\Sigma'_m| \geq n' - t'$ now "replaces" the sender's signature in the above broadcast protocol. Now we present the consensus protocol for $\mathcal{P}'$, each P_i holding a message $m_i \in \mathbb{F}$:

Protocol Consensus′**.**

0. $\forall P_i \in \mathcal{P}'$: Send m_i and the corresponding (alternative) signature σ'_i to all players in $\mathcal{P}'$.
1. $\forall P_i \in \mathcal{P}'$: If there exists a message m received (together with a valid signature) from at least $n'-t'$ different players, let Σ'_m denote the set of all these signatures, and set $\mathcal{A} = \mathcal{N} = \{m\}$ and $\Sigma_m = \{\}$. If no such message exists, set $\mathcal{A} = \mathcal{N} = \{\}$.

k. In each Step $k = 2, \ldots, t' + 2$, execute the following sub-steps for every player $P_i \in \mathcal{P}'$:

  $k.1$ For every message $m \in \mathcal{N}$, compute the signature σ_i on m, and send $(m, \Sigma'_m, \Sigma_m \cup \{\sigma_i\})$ to all players in $\mathcal{P}'$. Set $\mathcal{N} = \{\}$.

  $k.2$ In turn, for every message (m, Σ'_m, Σ_m) received in Substep $k.1$ do:

  - If $m \in \mathcal{A}$, or if $|\mathcal{A}| \geq 2$, ignore the message,
  - else if Σ_m contains valid signatures in the primary scheme from at least $k - 1$ different players in $\mathcal{P}'$, and Σ'_m contains valid signatures in the alternative scheme from at least $n' - t'$ different players in $\mathcal{P}'$, then include m in $\mathcal{A}$ and in $\mathcal{N}$,
  - else ignore the received message (m, Σ'_m, Σ_m) and all further messages from the player who has sent it.

$t'+3$. $\forall P_i$: if $|\mathcal{A}| = 1$, accept $m \in \mathcal{A}$ as the agreed value, otherwise (there was no pre-agreement) accept $m = \perp$.

The security of the protocol Consensus′ follows immediately from the security of the protocol Broadcast′, and the fact that every player issues at most one signature in the alternative scheme, and each such signature is verified at most $t+1$ times. The communication complexity of BA in $\mathcal{P}'$ is at most $4n^3|\sigma| + 3n^2\kappa + n^2|\sigma| = (8n^4 + 26n^3 + 9n^2)\kappa$.

Broadcast and consensus in $\mathcal{P}$ can be constructed from consensus in $\mathcal{P}'$:

Protocol Broadcast**.**

1. The sender $P_S \in \mathcal{P}$ sends the message m to every player $P_j \in \mathcal{P}'$.
2. Invoke Consensus′ to reach agreement on m among $\mathcal{P}'$.
3. Every player $P_i \in \mathcal{P}'$ sends the agreed message m to every player $P_j \in \mathcal{P}$.
4. Every player $P_j \in \mathcal{P}$ accepts the message m which was received most often.

Protocol Consensus**.**

1. Invoke Consensus′ to reach agreement on m among $\mathcal{P}'$.
2. Every player $P_i \in \mathcal{P}'$ sends the agreed message m to every player $P_j \in \mathcal{P}$.
3. Every player $P_j \in \mathcal{P}$ accepts the message m which was received most often.

The security of these protocols follows from the security of Consensus′ and from $t' < n'/2$ and $t < n/2$. The communication complexity of BA in $\mathcal{P}$ is at most $(8n^4 + 26n^3 + 11n^2)\kappa$.

6.6 Refreshing the Setup

6.6.1 Overview

To "refresh" the setup means to compute a new setup which allows for two BA operations, while this computation consumes only one BA-setup.

The protocol Refresh generates the new setup with a special-purpose MPC among the players in $\mathcal{P}'$.

This computation is performed *non-robustly*: The adversary can cause the generated setup to be incorrect, however this will be noticed by at least one honest player. More precisely: Every player has an internal state (the happy-bit), which is set to happy at the beginning of the computation. If a player detects a fault, he gets unhappy (sets his happy-bit to "unhappy"). We say that the protocol *succeeded* if all players remained happy, otherwise, the protocol *failed*. For the computation the following holds: If all players follow the protocol, then it succeeds (completeness) and if the protocol succeeds then it achieves its intended goal (correctness).

We do not require agreement on the fact whether or not a sub-protocol has failed. Only at the very end of Refresh, the players agree on whether or not a player has detected a failure during the computation (using consensus, thereby consuming one BA-setup). The computation takes only random values as input, so in case of failure, privacy is of no interest.

We provide a fault-handling sub-protocol, to be invoked when Refresh fails, which localizes a set $E \subseteq \mathcal{P}'$ of two players, where (at least) one of them is faulty. This allows to reduce the actual player set, thereby reducing the maximum number of faulty players, thereby limiting the number of times Refresh can fail. In this fault-handling sub-protocol, every players sends to some designated player all messages he has received during the course of the protocol, as well as all random elements he sampled (which define the sent messages). Given this information, the designated player can help to compute the set E to eliminate.

In the sequel, we present the used sub-protocols (all of them non-robust), and finally the protocols Refresh and FaultHandling. The protocol Refresh invokes once the protocol Consensus', hence it consumes one valid BA-setup. The protocol FaultHandling invokes 3 times the protocol Broadcast; it requires enough BA-setups for that. However, the protocol FaultHandling is invoked only t times in total, so the required BA-setups can be prepared at beforehand.

6.6.2 Secret Sharing

We define a correct d-sharing of a value as in Definition 2, i.e. we say that a value s is correctly d-shared among the players $\mathcal{P}'$ if there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$, and every (honest) player $P_i \in \mathcal{P}'$

holds a share $s_i = p(\alpha_i)$, where α_i is the unique evaluation point assigned to P_i. We denote the collection of shares as $[a]_d$.

As usual, by saying that the players in $\mathcal{P}$ compute (locally)

$$([y^{(1)}]_{d'}, \ldots, [y^{(m')}]_{d'}) = f([x^{(1)}]_d, \ldots, [x^{(m)}]_d)$$

(for any function $f : \mathbb{F}^m \to \mathbb{F}^{m'}$) we mean that every player P_i applies this function to his shares, i.e. computes

$$(y_i^{(1)}, \ldots, y_i^{(m')}) = f(x_i^{(1)}, \ldots, x_i^{(m)}).$$

Remember that by applying any linear or affine function to correct d-sharings we get a correct d-sharing of the output. However, by multiplying two correct d-sharings we get a correct $2d$-sharing of the product, i.e. $[a]_d[b]_d = [ab]_{2d}$.

As all sharings used in this chapter are t'-sharings or (temporarily) $2t'$-sharings, we denote the t'-sharing $[a]_{t'}$ as $[a]$ and the $2t'$-sharing $[a]_{2t'}$ as $[[a]]$.

In order to let a dealer $P_D \in \mathcal{P}'$ verifiably share a value a, we employ the following (non-robust) protocol (based on the VSS protocol of [BGW88])

Protocol Share**.**

1. DISTRIBUTION: P_D selects the coefficients $c_{0,1}, c_{1,0}, \ldots, c_{t',t'}$ at random, and sets $f(x, y) = a + c_{1,0}x + c_{0,1}y + c_{1,1}xy + \ldots + c_{t',t'}x^{t'}y^{t'}$. Then, to every $P_i \in \mathcal{P}'$, P_D computes and sends the polynomials $f_{i,\star}(y) = f(\alpha_i, y)$ and $f_{\star,i}(x) = f(x, \alpha_i)$.
2. CHECKING: For every pair $P_i, P_j \in \mathcal{P}'$, P_i sends $f_{i,\star}(\alpha_j)$ to P_j, who compares the received value with $f_{\star,j}(\alpha_i)$. P_j gets unhappy if some difference is non-zero.
3. OUTPUT: Every P_j outputs as his share $a_i = f_{i,\star}(0)$.

Lemma 24 *The protocol* Share *has the following properties: (Completeness) If all players in $\mathcal{P}'$ correctly follow the protocol, then the protocol succeeds. (Correctness) If the protocol succeeds, then the outputs $(a_1, \ldots, a_{n'})$ define a correct sharing of some value a', and for an honest dealer with input a it holds that $a = a'$. (Privacy) If the dealer is honest no subset of t' players obtains any information on his secret a.*

The protocol communicates at most $(2n^2 - 2n)\kappa$ bits and requires at most $(1/4 n^2 + 1/2 n - 3/4)\kappa$ random bits.

The following protocol lets the players in $\mathcal{P}'$ reconstruct a correctly shared value a towards a designated player $P_R \in \mathcal{P}'$:

Protocol Recons**.**

1. Every player $P_i \in \mathcal{P}'$ sends his share a_i to the recipient P_R.
2. P_R verifies whether $a_1, \dots, a_{n'}$ lie on a degree-t' polynomial $p(\cdot)$ and outputs $a = p(0)$ if yes. Otherwise, P_R gets unhappy and outputs $a = 0$.

Lemma 25 *The protocol* Recons *has the following properties: (Completeness) If all players in $\mathcal{P}'$ correctly follow the protocol, then the protocol succeeds. (Correctness) If the protocol succeeds, then P_R outputs the correct secret a. (Privacy) If the recipient is honest the adversary obtains no information on a.*

The protocol communicates at most $(n-1)\kappa$ bits and requires no randomness.

6.6.3 Generating Random Values

In the following, we present two (trivial) protocols for generating random values. The first one, GenerateRandom generates a correct sharing $[r]$ of a secret random value $r \in_R \mathbb{F}$ (or fails with at least one honest player being unhappy). The second one, GenerateRandomChallenge generates a random challenge $c \in_R \mathbb{F}$ known to all players in $\mathcal{P}'$ (or fails with at least one honest player being unhappy).

The protocol GenerateRandom non-robustly generates a secret random sharing by letting $t'+1$ players in $\mathcal{P}'$ verifiably share a random contribution and outputting the sum of this sharings.

Protocol GenerateRandom**.**

1. SHARE: $\forall P_i \in \{P_1, \dots, P_{t'+1}\}$: select a random value $c^{(i)} \in_R \mathbb{F}$ and invoke Share to share $c^{(i)}$ among $\mathcal{P}'$, resulting in $t'+1$ sharings $[c^{(1)}], \dots, [c^{(t'+1)}]$.
2. COMPUTE AND OUTPUT: The players compute (locally) and output the sharing $[c] = \sum_{i=1}^{t'+1} [c^{(i)}]$.

Lemma 26 *The protocol* GenerateRandom *has the following properties: (Completeness) If all players in $\mathcal{P}'$ correctly follow the protocol, then the protocol succeeds. (Correctness) If the protocol succeeds, then it generates a correct sharing $[r]$ of a uniformly random value $r \in_R \mathbb{F}$. (Privacy) Any subset of at most t' players has no joint information about r.*

The protocol communicates at most $(n^3 - n)\kappa$ bits and requires at most $(\frac{1}{8}n^3 + \frac{3}{8}n^2 + \frac{3}{8}n - \frac{5}{8})\kappa$ random bits.

Proof:[sketch] Completeness and complexity follow from inspecting the protocol. We now focus on the case when the protocol succeeds. There is at least one honest player P_h in $\{P_1, \ldots, P_{t'+1}\}$, who chooses his value $c^{(h)}$ uniformly at random. As in Step 1 the adversary does not obtain any information about $c^{(h)}$ (privacy of Share), and as the values $c^{(i)}$ of *every* player $P_i \in \mathcal{P}'$ are fixed after Step 1 (Correctness of Share), $c^{(h)}$ is statistically independent of all other values $c^{(j)}$ $(j \neq i)$. Hence, the sum $c^{(1)} + \ldots + c^{(t'+1)}$ is uniformly distributed and unknown to the adversary. ■

The following protocol GenerateRandomChallenge non-robustly generates a random challenge known to all players in $\mathcal{P}'$ by generating a random sharing and reconstructing it towards every player in $\mathcal{P}'$.

Protocol GenerateRandomChallenge**.**

1. GENERATE RANDOM SHARING $[c]$: Invoke GenerateRandom, to generate a sharing $[c]$ of a secret random value c.
3. RECONSTRUCT AND OUTPUT c: $\forall P_k \in \mathcal{P}'$: invoke Recons to reconstruct $[c]$ towards player P_k.

Lemma 27 *The protocol* GenerateRandomChallenge *has the following properties: (Completeness) If all players in $\mathcal{P}'$ correctly follow the protocol, then the protocol succeeds. (Correctness) If the protocol succeeds, then it generates a uniformly random value $c \in_R \mathbb{F}$, known to all players $P_j \in \mathcal{P}'$.*

The protocol communicates at most $(n^3 + n^2 - 2n)\kappa$ bits and requires at most $(\frac{1}{8}n^3 + \frac{3}{8}n^2 + \frac{3}{8}n - \frac{5}{8})\kappa$ random bits.

Proof: Follows immediately from the security of GenerateRandom and Recons. ■

6.6.4 Generating one Sig-setup

Recall that a sig-setup for a designated signer P_S consists of the signing key $(p_0, \ldots, p_{n'+1}, q_0, \ldots, q_{n'+1})$, which should be random and known only to the signer P_S, and one verification key $(v_{i,1}, \ldots, v_{i,n'+1}, x_i, y_i)$ for each player $P_i \in \mathcal{P}'$, where the values $v_{i,1}, \ldots, v_{i,n'+1}$ should be random and known only to P_i,[38] and the values x_i and y_i are computed as $x_i = p_0 + \sum_{j=1}^{n'+1} p_j v_{i,j}$ and $y_i = q_0 + \sum_{j=1}^{n'+1} q_j v_{i,j}$, respectively. Table 6.1 summarizes the steps needed to compute these values.

Player	Inputs (rand.)			Intermediate (shared)			Outputs
P_S	p_0	$\cdots$	$p_{n'+1}$				
	q_0	$\cdots$	$q_{n'+1}$				
P_1	$v_{1,1}$	$\cdots$	$v_{1,n'+1}$	$p_1 v_{1,1}$	$\cdots\cdots$	$p_{n'+1} v_{1,n'+1}$	$x_1 = p_0 + \sum_k p_k v_{1,k}$
				$q_1 v_{1,1}$	$\cdots\cdots$	$q_{n'+1} v_{1,n'+1}$	$y_1 = q_0 + \sum_k q_k v_{1,k}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots\cdots$
$P_{n'}$	$v_{n',1}$	$\cdots$	$v_{n',n'+1}$	$p_1 v_{n',1}$	$\cdots\cdots$	$p_{n'+1} v_{n',n'+1}$	$x_{n'} = p_0 + \sum_k p_k v_{n',k}$
				$q_1 v_{n',1}$	$\cdots\cdots$	$q_{n'+1} v_{n',n'+1}$	$y_{n'} = q_0 + \sum_k q_k v_{n',k}$

Table 6.1: Preparing one sig-setup

In our protocol, first every player P_i chooses and secret-shares his verification key $(v_{i,1}, \ldots, v_{i,n'+1})$. Then, the players jointly generate three random vectors $(p_0, \ldots, p_{n'+1})$, $(q_0, \ldots, q_{n'+1})$, and $(r_0, \ldots, r_{n'+1})$. The first two of these vectors will serve as signing key, and the third will serve as blinding in the verification of the computation. Then, for each of these three vectors, the values $x_1, \ldots, x_{n'}$, respectively $y_1, \ldots, y_{n'}$ or $z_1, \ldots, z_{n'}$, are computed. This computation is not detectable: It might be that one of the x_i, y_i or z_i values is wrong, and still no honest player has detected a failure (however, when all players correctly follow the protocol, then all values will be correct). The correctness of these values is verified in an additional verification step: Two random challenges ρ and φ are generated, and the linearly combined (and blinded) signing vector $(\rho p_0 + \varphi q_0 + r_0, \ldots, \rho p_{n'+1} + \varphi q_{n'+1} + r_{n'+1})$ is computed, and (distributively) compared with the linearly combined verification keys. If all

[38]The randomness of $v_{i,1}, \ldots, v_{i,n'+1}$ is needed for the sole reason of protecting the verifier P_i, hence it must be guaranteed for honest verifiers only.

checks are successful, then (with overwhelming probability) all keys are correctly computed.

Protocol GenerateSignatureSetup**.**

1. GENERATE $v_{i,k}$-VALUES: Every player $P_i \in \mathcal{P}'$ selects $n'+1$ random values $v_{i,1}, \ldots, v_{i,n'+1}$ and invokes Share to share them.
2. GENERATE p_k-VALUES: GenerateRandom is invoked $n'+1$ times to obtain random sharings $[p_0], \ldots, [p_{n'+1}]$.
3. COMPUTE x_i-VALUES: For every $i = 1, \ldots, n'$ the sharing $[x_i]$ of $x_i = p_0 + \sum_{j=1}^{n'+1} p_j v_{i,j}$ is computed as as follows:
   3.1 COMPUTE $[[x_i]]$: The players in $\mathcal{P}'$ (locally) compute the $2t'$-sharing $[[x_i]]$ of x_i as $[[x_i]] = [p_0] + \sum_{k=1}^{n'+1} [p_k][v_{i,k}]$.
   3.2 REDUCE DEGREE $[[x_i]] \rightarrow [x_i]$: Every player $P_j \in \mathcal{P}'$ shares his $2t'$-share $x_{i,j}$ of x_i acting as a dealer in Share, resulting in t'-sharings $[x_{i,1}], \ldots, [x_{i,n'}]$. The players compute (locally) the sharing $[x_i]$ of x_i as $[x_i] = \sum_{j=1}^{n'} \lambda_j [x_{i,j}]$, where λ_j denotes the j-th Lagrange coefficient, i.e. $\lambda_j = \prod_{i=1, i\neq j}^{n'} \frac{-\alpha_i}{\alpha_j - \alpha_i}$.
4. GENERATE q_k/y_i-VALUES: Generate $(q_0, \ldots, q_{n'+1})$ and $(y_1, \ldots, y_{n'})$ along the lines of Steps 2–3.
5. GENERATE r_k/z_i-VALUES: Generate $(r_0, \ldots, r_{n'+1})$ and $(z_1, \ldots, z_{n'})$ along the lines of Steps 2–3.
6. CHECK CORRECTNESS OF THE COMPUTED x_i/y_i-VALUES:
   6.1 Invoke GenerateRandomChallenge twice to generate random challenges ρ and φ.
   6.2 For $k = 1, \ldots, n'+1$, compute and reconstruct towards every player $[s_k] = \rho[p_k] + \varphi[q_k] + [r_k]$.
   6.3 For $i = 1, \ldots, n'$, compute $[w_i] = s_0 + \sum_{k=1}^{n'+1} s_k [v_{i,k}]$.
   6.4 For $i = 1, \ldots, n'$, compute $[\widetilde{w}_i] = \rho[x_i] + \varphi[y_i] + [z_i]$.
   6.5 For $i = 1, \ldots, n'$, reconstruct to every player $[d_i] = [w_i] - [\widetilde{w}_i]$.
   6.6 Every P_j checks whether $d_i \stackrel{?}{=} 0$ for $i = 1, \ldots, n'$, and gets unhappy in case of any non-zero value.
7. ANNOUNCE x_i/y_i-VALUES: For every $P_i \in \mathcal{P}'$, invoke Recons to reconstruct $[x_i]$ and $[y_i]$ towards P_i.

Lemma 28 *The protocol* GenerateSignatureSetup *has the following properties:*

(Completeness) If all players in $\mathcal{P}'$ correctly follow the protocol, then the protocol succeeds. (Correctness) If the protocol succeeds, then (with overwhelming probability) it generates a correct signature setup. (Privacy) If the protocol succeeds, then no subset of t' players obtains any information they are not allowed to obtain.

The protocol communicates at most $(11n^4 + n^3 - 2n^2 - 10n)\kappa$ bits and requires at most $(2n^4 + 4n^3 + 2n^2 + 3)\kappa$ random bits.

Proof:[sketch] (Completeness) We consider the case that all players follow the protocol, hence no sub-protocol fails. Observe that for every $i = 1, \dots, n'$, the points $(\alpha_1, x_{i,1}), \dots, (\alpha_{n'}, x_{i,n'})$ lie on a degree-$2t'$ polynomial $f_i(\cdot)$ with $f_i(0) = p_0 + \sum_{k=1}^{n'+1} p_k v_{i,k}$. This polynomial is well defined because $n' > 2t'$, hence we can interpolate $f_i(0)$ with Lagrange's formula.[39] This interpolation is done distributively, i.e., every player P_j shares his $x_{i,j}$, then these sharings are combined using Lagrange's formula, resulting in a sharing of $x_i = p_0 + \sum_{k=1}^{n'+1} p_k v_{i,k}$. Similarly, $y_i = q_0 + \sum_{k=1}^{n'+1} q_k v_{i,k}$ and $z_i = r_0 + \sum_{k=1}^{n'+1} r_k v_{i,k}$. Clearly, for any ρ and φ, $(\rho p_0 + \varphi q_0 + r_0) + \sum_{k=1}^{n'+1} (\rho p_k + \varphi q_k + r_k) v_{i,k} = \rho x_i + \varphi y_i + z_i$, hence $d_i = 0$, and no player detects a failure in Step 6.6.

(Correctness) We have to show that when the protocol succeeds, then for $i = 1, \dots, n'$ holds $x_i = p_0 + \sum_{k=1}^{n'+1} p_k v_{i,k}$ and $y_i = p_0 + \sum_{k=1}^{n'+1} q_k v_{i,k}$. Observe that after Step 5, the values $v_{i,k}, p_k, q_k, r_k, x_i, y_i, z_i$ are fixed (they all are t'-shared). When x_i and y_i do not satisfy the required equation above, then only with negligible probability, for random ρ and φ they satisfy the equation $(\rho p_0 + \varphi q_0 + r_0) + \sum_{k=1}^{n'+1} (\rho p_k + \varphi q_k + r_k) v_{i,k} = \rho x_i + \varphi y_i + z_i$.

(Privacy) We have to show that when the protocol succeeds, every player learns only his respective key (plus some random data he could have generated himself with the same probability distribution). First observe that in Steps 1–5, the only communication which takes place is by invocation of Share, which leaks no information to the adversary. In Step 6, the values $s_1, \dots, s_{n'+1}$ and $d_1, \dots, d_{n'}$ are reconstructed. Every value s_k is blinded with a random r_k (unknown to the adversary), so is uniformly random from the viewpoint of the adversary. The values d_i are either 0 (and hence the adversary can easily simulate them), or the protocol fails (and all computed values are discarded).

[39]Note that $f_i(0)$ is arbitrary when a single player is incorrect — something we do not care for when arguing about completeness.

The indicated complexities can be verified by inspecting the protocol.
■

6.6.5 Fault Detection

The following protocol FaultDetection enables the players in $\mathcal{P}'$ to agree on whether or not all players are happy.

Protocol FaultDetection**.**

1. DISTRIBUTE HAPPY-BITS: Every $P_i \in \mathcal{P}'$ sends his happy-bit to every $P_j \in \mathcal{P}'$, who gets unhappy if at least one P_i claims to be unhappy.
2. FIND AGREEMENT: The players in $\mathcal{P}'$ run Consensus′ with P_i's input being his happy-bit.
3. OUTPUT: If the consensus of the previous step outputs "happy", output "succeeded" otherwise output "failed".

Note that if at least one honest player inputs "unhappy", then FaultDetection outputs failed (regardless of the behavior of the adversary). If all honest players input "happy" and all players follow the protocol then FaultDetection outputs "succeeded". However if all honest players input "happy" the adversary can still cause the output to be "failed".

6.6.6 The Refresh Protocol

In order to refresh a BA-setup, we need to generate two BA-setups, consuming only one BA-setup. Remember that one BA-setup consists of $2n'$ sig-setups (2 for every potential signer); hence, Refresh needs to generate $4n'$ sig-setups.

Protocol Refresh**.**

0. Every $P_i \in \mathcal{P}'$ sets his happy-bit to happy.
1. Invoke GenerateSignatureSetup $4n'$ times in parallel to generate 4 sig-setups for each signer $P_S \in \mathcal{P}'$.
2. Invoke FaultDetection to agree on whether or not some player has detected a fault.

3. Every $P_i \in \mathcal{P}'$ sends the output of FaultDetection to every $P_j \in (\mathcal{P} \backslash \mathcal{P}')$, who outputs the message received most often.

It is easy to see that Refresh fails (every honest player outputs fail) when any GenerateSignatureSetup failed for an honest player. On the other hand, when all players follow the protocol (inclusive FaultDetection), then Refresh succeeds. Refresh communicates $\mathcal{O}(n^5)\kappa$ bits.

6.6.7 Fault Handling

The following fault-handling procedure is invoked only when Refresh has failed. The goal of FaultHandling is to localize a set $E \in \mathcal{P}'$ of two players, such that (at least) one of them is faulty.

FaultHandling exploits the fact that there is no need to maintain the secrecy of the failed Refresh protocol. Basically, in FaultHandling the whole transcript of Refresh is revealed and there will be a message from some player P_i to some player P_j, where P_i claims to have sent some other message than P_j claims to have received — hence either P_i or P_j is lying, and we can set $E = \{P_i, P_j\}$. Unfortunately, it would be too expensive to publicly reveal the whole transcript; instead, the transcript is revealed towards a selected player (e.g. $P_k \in \mathcal{P}'$ with the smallest index k), who searches for the fault and announces it.

We stress that the considered transcript not only contains the messages of all invocations of the protocol GenerateSignatureSetup, but also the messages of the protocol Refresh. This is important because it might be that no fault occurred in GenerateSignatureSetup, but still some (corrupted) player P_i claimed to be unhappy in FaultDetection.

Protocol FaultHandling**.**

1. Every $P_i \in \mathcal{P}'$ sends to P_k all random values chosen during the course of the protocol Refresh (including all sub-protocols), as well as all values received during the course of Refresh.
2. P_k computes for every P_i the messages P_i should have sent (when being correct) during the course of Refresh; this can be done based on the random values and the received messages of P_i.
3. P_k searches for a message from some player $P_i \in \mathcal{P}'$ to some other player $P_j \in \mathcal{P}'$, where P_i should have sent a message x_i (according to his claimed randomness), but P_j claims to have received x_j, where $x_i \neq x_j$. Denote the index of this message by ℓ.

4. P_k invokes Broadcast to announce (i, j, ℓ, x_i, x_j).
5. P_i invokes Broadcast to announce whether he indeed sent x_i in the ℓ-th message.
6. P_j invokes Broadcast to announce whether he indeed received x_j in the ℓ-th message.
7. If Both P_i and P_j confirm to have sent x_i, respectively to have received x_j, then $E = \{P_i, P_j\}$. If P_i does not confirm to have sent x_i, then $E = \{P_k, P_i\}$. If P_j does not confirm to have received x_j, then $E = \{P_k, P_j\}$.

FaultHandling requires 3 BA invocations and communicates $O(n^5\kappa)$ bits.

6.7 Long Messages

The proposed BA protocols only capture messages $m \in \mathbb{F}$, i.e., κ-bit messages. In order to reach BA on longer messages, one could invoke the according BA protocol several times (once for every κ bit block). However, this would blow up the communication complexity unnecessarily high: BA of a $\ell\kappa$ bit message would require a communication complexity of $\mathcal{O}(\ell n^5\kappa)$ bits (as opposed to $\mathcal{O}(\ell\kappa n^2 + n^{17}\kappa)$ in [PW96]). In this section, we sketch a construction that allows BA of a $\ell\kappa$ bit message at costs of $\mathcal{O}(\ell\kappa n^2 + n^5\kappa)$ bits.

In order to achieve the stated complexity, we need to replace the protocol Consensus′ by Consensus$_{\text{long}}$′. The basic idea of Consensus$_{\text{long}}$′ is straight forward: Every player $P_i \in \mathcal{P}'$ sends his message m_i to every other player. Then, the players use Consensus′ to reach agreement on a universal hash value. If agreement is achieved, all players output the message with the agreed hash value, otherwise they output $\perp$. The key for the universal hash function is assumed to be pre-shared among the players as part of the BA-setup, and only reconstructed when needed. We also explain how this sharing is prepared in the Refresh protocol.

6.7.1 Protocol Consensus$_{\text{long}}$′

In the following, we present the protocol Consensus$_{\text{long}}$ among the players in $\mathcal{P}'$, reaching agreement on a $\ell\kappa$ bit message m. The protocol makes use of universal hashing [CW79]. As universal hash with key $k \in \mathbb{F}$, we use the function $U_k : \mathbb{F}^\ell \to \mathbb{F}, (m^{(1)}, \ldots, m^{(\ell)}) \mapsto m^{(1)} + m^{(2)}k + \ldots + m^{(\ell)}k^{\ell-1}$.

The probability that two different messages map to the same hash value for a uniformly chosen key is at most $\ell/|\mathbb{F}|$, which is negligible in our setting with $\mathbb{F} = \mathrm{GF}(2^\kappa)$.

Protocol $\mathsf{Consensus}_{\mathrm{long}}{}'$**.**

1. Every $P_i \in \mathcal{P}'$ sends his message m_i to every player $P_j \in \mathcal{P}'$.
2. The players reconstruct the random hash key $k \in \mathbb{F}$, which is part of the BA setup.
3. Every $P_i \in \mathcal{P}'$ computes (for his original message m_i) the universal hash $U_k(m_i)$.[40]
4. The players in $\mathcal{P}'$ invoke $\mathsf{Consensus}'$ to reach agreement on the hash value h.
5. If the above consensus fails (i.e., $h = \perp$), then every $P_i \in \mathcal{P}'$ outputs $\perp$. If it succeeds, then every $P_i \in \mathcal{P}'$ outputs that m_j received in Step 1 with $U_k(m_j) = h$.

One can easily see that the above protocol reaches consensus on m, and that it communicates $\mathcal{O}(\ell\kappa n^2)$ plus one invocations of $\mathsf{Consensus}'$, i.e., communicates $\mathcal{O}(\ell\kappa n^2 + n^4\kappa)$ overall.

6.7.2 Generating the Hash Key

The protocol $\mathsf{Consensus}_{\mathrm{long}}{}'$ needs a random hash key to be known to all players in $\mathcal{P}'$. We cannot afford to generate this hash key on-line (this would require several invocations of broadcast). Instead, we assume a robust sharing of a random field element to be part of every BA-setup. This sharing is then reconstructed when needed.

As robust sharing, we use the scheme of [CDD$^+$99]. Essentially, this is a two-dimensional Shamir sharing, ameliorated with so called authentication tags. The sharing is constructed non-robustly; in the Share protocol, the players pair-wisely check the consistency of the received shares, and fail in presence of faults. The sharing of the hash key is generated as sum of a sharing of each player in $\mathcal{P}'$. Such a sharing can be computed with communicating $\mathcal{O}(n^4\kappa)$ bits (and without involving broadcast). When the hash key is needed, then the sharing of the actual BA setup is reconstructed towards every player in $\mathcal{P}'$. This is achieved

[40]In order to do so, the message m_i is split into blocks $m_i^{(1)}, \ldots, m_i^{(\ell)}$.

by having every player sending his shares (including the authentication tags) to every other player; this involves a communication of $\mathcal{O}(n^3\kappa)$ bits.

Chapter 7

Asynchronous MPC

7.1 Introduction

Up to now, we have considered computations in synchronous networks only. In this chapter we concentrate on asynchronous communication which models real-world networks (like the Internet) much better than synchronous communication.

Remember that in asynchronous networks, messages are delayed arbitrarily (as a worst-case assumption, the adversary is given the power to schedule the delivery of messages). Thus when a player does not receive an expected message, he cannot decide whether the sender is corrupted (and did not send the message at all) or the message is just delayed in the network. This makes protocols for asynchronous networks much more involved than their synchronous counterparts. It also implies that in asynchronous settings it is impossible to consider the inputs of *all* uncorrupted players. The inputs of up to t (potentially honest) players have to be ignored, because waiting for them could turn out to be endless.

For a good introduction to asynchronous protocols, see [Can95]. Due to its complexity, asynchronous MPC has attracted much less research than synchronous MPC. The most important results on asynchronous MPC are [BCG93, BKR94, SR00, PSR02, HNP05, HNP08].

In this chapter we present an MPC protocol providing perfect security against an active adaptive adversary corrupting up to $t < n/4$ of the players (which is optimal) and communicating $O(n^3\kappa)$ bits per multiplication. Furthermore, we extend the protocol for a hybrid communication model (with the same security properties and the same communication complexity), allowing *all* players to give input if the *very first communication round* is synchronous, and taking at least $n - t$ inputs in a fully asynchronous setting. More precisely, the extended protocol takes the inputs of at least $n - t$ players, and additionally, always takes the inputs of players whose first-round messages are delivered within some a-priory fixed time.

The results presented in this chapter were published in [BH07].

7.2 Model

We consider a set $\mathcal{P}$ of n players, $\mathcal{P} = \{P_1, \ldots, P_n\}$, connected with a complete network of secure (private and authentic) asynchronous channels. The function to be computed is specified as an arithmetic circuit over a finite field $\mathbb{F}$ (with $|\mathbb{F}| > n$), with input, addition, multiplication, random, and output gates. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O, respectively.

The faultiness of players is modeled in terms of a central adversary corrupting players. The adversary can corrupt up to t players for any fixed t with $t < n/4$, and make them deviate from the protocol in any desired manner. The adversary is computationally unbounded, active, and adaptive. Furthermore, the adversary can schedule the delivery of the messages in the network, i.e., she can delay any message arbitrarily. In particular, the order of the messages does not have to be preserved. However, every sent message will eventually be delivered.

The security of our protocols is perfect, i.e., information-theoretic without any error probability.

7.3 Preliminaries

7.3.1 Design of Asynchronous MPC Protocols

Asynchronous protocols are executed in *steps*. Each step begins by the scheduler choosing one message (out of the queue) to be delivered to its designated recipient. The recipient is activated by receiving the message, he performs some (internal) computation and possibly sends messages on his outgoing channel (and waits for the next message).

The action to be taken by the recipient is defined by the relevant sub-protocol[41] consisting of a number of instructions what is to be done upon receiving a specified message. If the received message refers to a sub-protocol which is not yet "in execution", then the player keeps the message until the relevant sub-protocol is invoked.

7.3.2 Partial Termination

Many "asymmetric" tasks with a designated dealer (broadcast, secret-sharing) cannot be implemented with guaranteed termination in an asynchronous world; the players cannot distinguish whether the dealer is corrupted and does not start the protocol, or the dealer is correct but his messages are delayed in the network. Hence, these protocol are required to terminate only if the dealer is correct. However, we require that if such a sub-protocol terminated for one (correct) player, then it must eventually terminate for all correct players.

The issue with partial termination is typically attacked by invoking n instances of the protocol with partial termination in parallel, every player acting as dealer in one instance. Then, every player can wait until $n-t$ instances have terminated (from his point of view). In order to reach agreement on the set of terminated instances, a specialized sub-protocol is invoked, called agreement on a core-set. A player can only be contained in the core-set if his protocol instance has terminated for at least one honest player, and hence will eventually terminate for all honest players. The core-set contains at least $n - t$ players.

7.3.3 Input Provision

Providing input is an inherently asymmetric task, and it is not possible to distinguish between a corrupted input player who does not send any message and a correct input player whose messages are delayed in the network. For this reason, in a fully asynchronous world it is not possible to take the inputs of *all* players; up to t (possible correct) players cannot be waited for, as this waiting could turn out to be endless. Hence, the protocol waits only till $n-t$ of the players have achieved to provide input, and then goes on with the computation.

[41]We assume that for each message it is clear which sub-protocol it belongs to.

7.3.4 Byzantine Agreement

We need three flavors of Byzantine agreement, namely broadcast, consensus, and core-set agreement.

The broadcast (BC) primitive allows a sender to distribute a message among the players such that all players get the same message (even when the sender is corrupted), and the message they get is the sender's message if he is honest. As explained above, broadcast cannot be realized with complete termination; instead, termination of all (correct) players is required only when the sender is correct; however, as soon as at least one correct player terminates, all players must eventually terminate. Such a broadcast primitive can be realized rather easily [Bra84]. The required communication for broadcasting an ℓ-bit message is $\mathcal{O}(n^2\ell)$, where the hidden constant is small.

Consensus enables a set of players to agree on a value. If all honest players start the consensus protocol with the same input value v then all honest players will eventually terminate the protocol with the same value v as output. If they start with different input values, then they will eventually reach agreement on some value. All known i.t.-secure asynchronous consensus protocols start by having every player broadcast his input value, which results in communication complexity $\Omega(n^3\ell)$, where ℓ denotes the length of the inputs.

Agreement on a core set (ACS) is a primitive introduced in [BCG93] (a more efficient ACS protocol can be found in [BKR94]). We use it to determine a set of at least $n - t$ players that correctly shared their values. More concretely, every player starts the ACS protocol with an accumulative set of players who from his point of view correctly shared one or more values (the share sub-protocol in which they acted as dealers terminated properly). The output of the protocol is a set of at least $n-t$ players, who indeed correctly shared their values, which means that every honest player will eventually get a share of every sharing dealt by a dealer from the core set. The communication costs of an ACS protocol are essentially the costs of n invocations of bit-consensus, i.e. $\Omega(n^4)$ bits.

7.4 Protocol Overview

Our asynchronous MPC protocol is very simple.

All protocols and sub-protocol are robust (we don't use player elimination or dispute control).

As sharing, we use the standard Shamir sharing, to share and reconstruct values we use the AVSS scheme from [BCG93] (extended to share many values in parallel). The share protocol produces correct t-sharings, whereas the reconstruct protocol reconstructs sharings up to degree $2t$.[42]

Due to the linearity of Shamir sharing, all linear and affine functions of shared values can be computed locally, without communication.

Random sharings are computed using super-invertible matrices: Every player verifiably shares his contribution(s), then a core set agreement is run, to agree on a set of at least $n - t$ players who correctly shared their values. The random sharings are then defined by applying a super-invertible matrix to these (correctly shared) contributions.

To multiply two shared values, we use a variation of a protocol from [DN07] (described in the Chapters 3 and 4), based on a double-sharing of a random value. As we cannot $2t$-share values directly with our share protocol ($2t > n/4$), we generate a $(t, 2t)$-sharing from $3t + 1$ random t-sharings.

The main protocol proceeds in three phases: the preparation phase, the input phase, and the computation phase. Every honest player will eventually complete every phase.

In the preparation phase many sharings of random values will be generated in parallel (using only one ACS). For every multiplication gate, $3t + 1$ random sharing will be generated. For every random gate, one random sharing will be generated.

In the input phase the players share their inputs and agree on a core set of correctly shared inputs. (Every honest player will eventually get a share of every input from the core set.)

In the computation phase, the actual circuit will be computed gate by gate, based on the core-set inputs. Due to the linearity of the used secret-sharing, the linear gates can be computed locally – without communication. Each multiplication gate will be evaluated with the help of $3t + 1$ prepared random sharings. First a random double-sharing will be generated (from the $3t + 1$ sharings), which will then be used to multiply the two factors. The output gates are evaluated with a robust reconstruct protocol.

[42]The reconstruct protocol works for degree $d < n/2$ whereas the share protocol requires $d < n/4$

7.5 Secret Sharing

7.5.1 Definitions and Notations

We define a correct d-sharing of a value according to Definition 4: We say that a value s is *d-shared* if every correct player P_i is holding a share s_i of s, such that there exists a degree-d polynomial $p(x)$ with $p(0) = s$ and $p(\alpha_i) = s_i$ for all $i = 1, \ldots, n$. We call the vector $(s_1, \ldots, s_n)$ of shares a d-sharing of s. A (possibly incomplete) set of shares is called *d-consistent* if these shares lie on a degree d polynomial.

Most of our sharings will be t-sharings (where t denotes the maximum number of corrupted players). We denote a t-sharing of s by $[s]$. In the multiplication sub-protocol, we will also use $2t$-sharings, which will be denoted by $[[s]]$.

7.5.2 Share$_1$ and Recons— The Vanilla Protocols

In the following, we recap the asynchronous verifiable secret-sharing scheme of [BCG93], consisting of the protocols Share$_1$ and Recons.[43] Share$_1$ allows a dealer P_D to verifiably t-share a secret value $s \in \mathbb{F}$. Recons allows the players to reconstruct a d-sharing (for $d \leq 2t$) towards a receiver P_R. We stress that the protocol Share$_1$ does not necessarily terminate when the dealer P_D is corrupted. However, when it terminates for some correct player, then it eventually terminates for all players. The protocol Recons always terminates.

The intuition behind the protocol Share$_1$ is the following: In order to share a secret s, the dealer chooses a random two-dimensional polynomial $f(\cdot,\cdot)$ with $f(0,0) = s$, and sends to every player P_i the polynomials $g_i(\cdot) = f(\alpha_i,\cdot)$ and $h_i(\cdot) = f(\cdot,\alpha_i)$. Then the players pairwise check the consistency of the received polynomials, and publicly confirm successful checks. Once $n - t$ players are mutually consistent, every other player P_i uses the checking points received from these players to determine his polynomial $g_i(\cdot)$, and computes his share $s_i = g_i(0)$.

Protocol Share$_1$ **(Dealer P_D, secret $s \in \mathbb{F}$).**

[43] We denote their sharing protocol by Share$_1$, as it allows to share only one single value.

- DISTRIBUTION — CODE FOR DEALER P_D: Choose a random two-dimensional degree-t polynomial $f(\cdot,\cdot)$ with $f(0,0) = s$, and send to each player P_i the two degree-t polynomials $g_i(\cdot) = f(\alpha_i,\cdot)$ and $h_i(\cdot) = f(\cdot,\alpha_i)$.
- CONSISTENCY CHECKS — CODE FOR PLAYER P_i:
  1. Wait for $g_i(\cdot)$ and $h_i(\cdot)$ from P_D.
  2. To each player P_j send the share-share $s_{ji} = h_i(\alpha_j)$.
  3. Upon receiving s_{ij} from P_j check whether $s_{ij} = g_i(\alpha_j)$. If so broadcast (ok, i, j).
- OUTPUT-COMPUTING — CODE FOR PLAYER P_i:
  1. Wait until there is an $(n-t)$-clique in the graph implicitly defined by the broadcasted confirmations.[44]
  2. Upon receiving at least $2t+1$ t-consistent share-shares s_{ij} (for $j \in \{1,\ldots,n\}$) from the players in the clique, find the interpolation polynomial $\widetilde{g}_i(\cdot)$ and (re)compute your share $s_i = \widetilde{g}_i(0)$.[45]
  3. Output the share s_i.

Lemma 29 *For every coalition of up to t bad players and every scheduler, the protocol* Share_1 *achieves the following properties:*

- *Termination: If the dealer is correct, then every correct player will eventually complete* Share_1*. If some correct player has completed* Share_1*, then all the correct players will eventually complete* Share_1*.*
- *Correctness: Once a correct player has completed* Share_1*, there exists a unique value r which is correctly t-shared among the players where $r = s$ if the dealer is correct.*
- *Privacy: If the dealer is correct, then the adversary obtains no information about the shared secret.*

The communication complexity of Share_1 *is $\mathcal{O}(n^2\kappa + n^2\mathcal{BC}(\kappa))$.*

The intuition behind the protocol Recons is the following: Every player P_i sends his share s_i to P_R. The receiver waits until receiving at

[44]The graph has n nodes representing the n players and there is an edge between i and j if and only if both (ok, i, j) and (ok, j, i) were broadcasted.
[45]If the dealer is correct or if P_i is a member of the clique, then $g_i(\cdot) = \widetilde{g}_i(\cdot)$

least $d+t+1$ d-consistent shares and outputs the value of their interpolation polynomial at 0. Note that corrupted players can send false shares to P_R, but at the latest when P_R has received the shares of all honest players, he has at least $n-t \geq d+t+1$ d-consistent shares (for $t < n/4$ and $d \leq 2t$).

Protocol Recons **(Receiver P_R, degree d, d-sharing of s).**

- CODE FOR PLAYER P_i: Send s_i to P_R.
- CODE FOR RECEIVER P_R: Upon receiving at least $d+t+1$ d-consistent shares s_i (and up to t inconsistent shares), interpolate the polynomial $p(\cdot)$ and output $s = p(0)$.

Lemma 30 *For any d-shared value s, where $d + 2t < n$, for every coalition of up to t bad players, and for every scheduler, the protocol* Recons *achieves the following properties:*

- *Termination: Every correct player will eventually complete* Recons*.*
- *Correctness: P_R will output s.*
- *Privacy: When P_R is honest, then the adversary obtains no information about the shared secret.*

The communication complexity of the protocol Recons *is $\mathcal{O}(n\kappa)$.*

Note that for $t < n/4$, Recons can be used to reconstruct t-sharings as well as $2t$-sharings. However, the protocol Share$_1$ can only generate t-sharings.

Proofs of security as well as details on solving the clique-problem in Share$_1$ (respectively, reducing it to a computationally simpler problem) and on finding (and interpolating) $d+t+1$ d-consistent shares in Recons, can be found in [BCG93].

7.5.3 Share*: Sharing Many Values at Once

The following protocol Share* extends the protocol Share$_1$ in two ways: First, it allows the dealer to share a vector $\left(s^{(1)}, \ldots, s^{(\ell)}\right)$ of ℓ secrets at once, substantially more efficiently than ℓ independent invocations of Share$_1$. Secondly, Share* allows to share "empty" secrets, formally denoted as $s^{(k)} = \perp$, resulting in all shares of $s^{(k)}$ being $\perp$ as well. This will be used when a dealer should share an unknown value.

Protocol Share* **(Dealer P_D, secrets $(s^{(1)},\dots,s^{(\ell)}) \in (\mathbb{F}\cup\{\perp\})^\ell$).**

- DISTRIBUTION — CODE FOR DEALER P_D: For every $s^{(k)} \neq \perp$, choose a random two-dimensional degree-t polynomial $f^{(k)}(\cdot,\cdot)$ with $f^{(k)}(0,0) = s^{(k)}$. Send to every P_i the polynomials $\big(g_i^{(1)}(\cdot),h_i^{(1)}(\cdot),\dots,g_i^{(\ell)}(\cdot),h_i^{(\ell)}(\cdot)\big)$, where $g_i^{(k)}(\cdot) = f^{(k)}(\alpha_i,\cdot)$ and $h_i^{(k)}(\cdot) = f^{(k)}(\cdot,\alpha_i)$ if $s^{(k)} \in \mathbb{F}$, and $g_i^{(k)} = h_i^{(k)} = \perp$ if $s^{(k)} = \perp$.
- CONSISTENCY CHECKS — CODE FOR PLAYER P_i:
  1. Wait for $\big(g_i^{(1)}(\cdot),h_i^{(1)}(\cdot),\dots,g_i^{(\ell)}(\cdot),h_i^{(\ell)}(\cdot)\big)$ from P_D.
  2. To each P_j send $\big(s_{ji}^{(1)},\dots,s_{ji}^{(\ell)}\big)$, where $s_{ji}^{(k)} = h_i^{(k)}(\alpha_j)$, resp. $s_{ji}^{(k)} = \perp$ if $h_i^{(k)} = \perp$.
  3. Upon receiving $\big(s_{ij}^{(1)},\dots,s_{ij}^{(\ell)}\big)$ from P_j, broadcast (ok,i,j) if for all $k = 1,\dots,\ell$ it holds that $s_{ij}^{(k)} = g_i^{(k)}(\alpha_j)$, resp. $s_{ij}^{(k)} = \perp = g_i^{(k)}$.
- OUTPUT-COMPUTING — CODE FOR PLAYER P_i:
  1. Wait until there is a $(n-t)$-clique in the graph defined by the broadcasted confirmations.
  2. For $k = 1,\dots,\ell$, upon receiving at least $2t+1$ t-consistent share-shares $s_{ij}^{(k)}$ (for $j \in \{1,\dots,n\}$) from the players in the clique, find the interpolation polynomial $\widetilde{g}_i^{(k)}(\cdot)$ and (re)compute the share $s_i^{(k)} = \widetilde{g}_i^{(k)}(0)$. Upon receiving $2t+1$ values $s_{ij}^{(k)} = \perp$ (for $j \in \{1,\dots,n\}$), set $s_i^{(k)} = \perp$.
  3. Output the shares $\big(s_i^{(1)},\dots,s_i^{(\ell)}\big)$.

Lemma 31 *The protocol* Share* *allows P_D to share ℓ secrets from $\mathbb{F}\cup\{\perp\}$ at once, with the same security properties as guaranteed in Lemma 29. The communication complexity of* Share* *is $\mathcal{O}(\ell n^2\kappa + n^2\mathcal{BC}(\kappa))$.*

7.6 Preparation Phase

The goal of the preparation phase is to generate t-sharings of ℓ uniformly random values $r^{(1)},\dots,r^{(\ell)}$, unknown to the adversary, where ℓ will be $c_M(3t+1)+c_R$.

The idea of the protocol PreparationPhase is the following: First, every player acts as dealer in Share* to share a vector of $\ell' = \lceil \ell/(n-2t)\rceil$

random values. Then the players agree on a core set of $n-t$ correct dealers (such that their Share* protocol was completed by at least one honest player). This results in $n-t$ vectors of ℓ' correct t-sharings, but up to t of these vectors may be known to the adversary. Then, these $n-t$ correct vectors are compressed to $n-2t$ correct *random* vectors, unknown to the adversary, by using a $(n-2t)$-by-$(n-t)$ super-invertible matrix (applied component-wise). This computation is linear, hence the players can compute their shares of the compressed sharings locally from their shares of the original sharings.

Protocol PreparationPhase **(ℓ).**

Code for player P_i:

- SECRET SHARING
  - Act as a dealer in Share* to share a vector of $\ell' = \lceil \ell/(n-2t) \rceil$ random values $\left(s^{(i,1)}, \ldots, s^{(i,\ell')}\right)$.
  - For every $j = 1, \ldots, n$, take part in Share* with dealer P_j, resulting in the shares $\left(s_i^{(j,1)}, \ldots, s_i^{(j,\ell')}\right)$.
- AGREEMENT ON A CORE SET
  1. Create an accumulative set $C_i = \emptyset$.
  2. Upon completing Share* with dealer P_j, include P_j in C_i.
  3. Take part in ACS with the accumulative set C_i as input.
- COMPUTE OUTPUT (LOCAL COMPUTATION)
  1. Wait until ACS completes with output C. For simple notation, assume that $\{P_1, \ldots, P_{n-t}\} \subseteq C$.
  2. For every $k \in \{1, \ldots, \ell'\}$, the $(n-2t)$ t-shared random values, unknown to the adversary, are defined as $\left(r^{(1,k)}, \ldots, r^{(n-2t,k)}\right) = M\left(s^{(1,k)}, \ldots, s^{(n-t,k)}\right)$, where M denotes a $(n-2t)$-by-$(n-t)$ super-invertible matrix. Compute your shares $\left(r_i^{(1,k)}, \ldots, r_i^{(n-2t,k)}\right)$ accordingly. Denote the first ℓ resulting sharings as $[r^{(1)}], \ldots, [r^{(\ell)}]$ (it holds that $\ell'(n-2t) \geq \ell$).

Lemma 32 PreparationPhase *(eventually) terminates for every honest player. It outputs independent random sharings of ℓ secret, independent, uniformly random values $r^{(1)}, \ldots, r^{(\ell)}$.* PreparationPhase *communicates $\mathcal{O}(\ell n^2 \kappa + n^3 \mathcal{BC}(\kappa))$ bits and requires one invocation of ACS.*

7.7 Input Phase

In the InputPhase protocol every player P_i acts as a dealer in one Share* protocol in order to share his input s_i.[46] However the asynchronity of the network does not allow the players to wait for more than $n - t$ Share*-protocols to be completed. In order to agree on a set of players whose inputs will be taken into to computation one ACS protocol is run.

Protocol InputPhase **(every P_i has input s_i).**

Code for player P_i:

- SECRET SHARING
  - Share your secret input s_i with Share*.
  - For every $j = 1, \dots, n$ take part in Share* with dealer P_j.
- AGREEMENT ON A CORE SET
  1. Create a accumulative set $C_i = \emptyset$.
  2. Upon completing Share* with dealer P_j, include P_j in C_i.
  3. Take part in ACS with your accumulative set C_i as your input.
  4. Output the agreed core set C and your outputs of the Share* protocols with dealers from C.

Lemma 33 *The* InputPhase *protocol will (eventually) terminate for every honest player. It enables the players to agree on a core set of at least $n-t$ players who correctly shared their inputs – every honest player will (eventually) complete the* Share* *protocol of every dealer from the core set (and get the correct shares of his shared input values).* InputPhase *communicates $\mathcal{O}(c_I n^2 \kappa + n^3 \mathcal{BC}(\kappa))$ bits and requires one invocation of* ACS.

7.8 Computation Phase

In the computation phase, the circuit is evaluated gate by gate, whereby all inputs and intermediate values are shared among the players. As soon as a player holds his shares of the input values of a gate, he joins the computation of this gate.

[46] s_i can be one value or an arbitrary long vector of values from $\mathbb{F}$

Due to the linearity of the secret-sharing scheme, linear gates can be computed locally simply by applying the linear function to the shares, i.e. for any linear function $f(\cdot,\cdot)$, a sharing $[c] = [f(a,b)]$ is computed by letting every player P_i compute $c_i = f(a_i, b_i)$. With every random gate, one random sharing (from the preparation phase) is associated, which is directly used as outcome of the random gate. With every multiplication gate, $3t+1$ random sharings (from the preparation phase) are associated, which are used to compute a sharing of the product as described in the protocol Multiplication.

Protocol ComputationPhase **(**$\ell = (3t+1)c_M + c_R$ **random sharings** $[r^{(1)}], \ldots, [r^{(\ell)}]$**).**

For every gate in the circuit — Code for player P_i:

1. Wait until you have shares of each of the inputs
2. Depending on the type of the gate, proceed as follows:
   - Linear gate $[c] = f([a],[b],\ldots)$: compute your share c_i as $c_i = f(a_i, b_i, \ldots)$.
   - Multiplication gate $[c] = [a][b]$: participate in protocol Multiplication$([a],[b],[r^{(0)}],\ldots,[r^{(3t)}])$, where $[r^{(0)}],\ldots,[r^{(3t)}]$ denote the $3t+1$ associated random sharings.
   - Random gate $[r]$: set your share $r_i = r_i^{(k)}$, where $[r^{(k)}]$ denotes the associated random sharing.
   - Output gate $[a] \to P_R$: participate in Recons$(P_R, d = t, [a])$.

In order to compute multiplication gates, we use the approach of of [DN07]: First, the players jointly generate a secret random value s, which is both t-shared (by $[s]$) and $2t$-shared (by $[[s]]$). These sharings can easily be generated based on the $3t+1$ t-sharings associated with the multiplication gate. Then, every player locally multiplies his shares of a and b, resulting in a $2t$-sharing of the product $c = ab$, i.e., $[[c]]$. Then, the players compute and reconstruct $[[c-s]]$, resulting in every player knowing $\delta = c - s$ and (locally) compute $[c] = \delta + [s]$, the correct product $[ab]$.

Protocol Multiplication **(**$[a],[b],[r^{(0)}],\ldots,[r^{(3t)}]$**).**

Code for player P_i:

1. PREPARE $[s]$: The degree-t polynomial $p(\cdot)$ to share s is defined by the shared coefficients $r^{(0)}, r^{(1)}, \ldots, r^{(t)}$. For every P_j, a sharing of his share $s_j = p(\alpha_j)$ is defined as $[s_j] = [r^{(0)}] + [r^{(1)}]\alpha_j + \ldots + [r^{(t)}]\alpha_j^t$. Participate in Recons$(P_j, d = t, [s_j])$ to let P_j learn his degree-t share s_j, resulting in $[s]$.
2. PREPARE $[[s]]$: The degree-$2t$ polynomial $p'(\cdot)$ to share s is defined by the shared coefficients $r^{(0)}, r^{(t+1)}, \ldots, r^{(3t)}$. For every P_j, a sharing of his share $s'_j = p'(\alpha_j)$ is defined as $[s'_j] = [r^{(0)}] + [r^{(t+1)}]\alpha_j + \ldots + [r^{(3t)}]\alpha_j^{2t}$. Participate in Recons$(P_j, d = t, [s'_j])$ to let P_j learn his degree-$2t$ share s'_j, resulting in $[[s]]$.
3. COMPUTE AND OUTPUT $[ab]$:
   1. Compute your degree-$2t$ share of $c = ab$ as $c_i = a_i b_i$, resulting in $[[c]] = [a][b]$.
   2. For every $j = 1, \ldots, n$, participate in Recons (P_j, $d = 2t$, ($[[c]] - [[s]]$)), resulting in every P_j knowing $\delta = c - s$.
   3. Compute and output your share c_i of $c = \delta + s$ as $c_i = \delta + s_i$, resulting in $[c] = [ab]$.

Lemma 34 *The protocol* Multiplication *(eventually) terminates for every honest player. Given correct sharings* $[a], [b], [r^{(0)}], \ldots, [r^{(3t+1)}]$ *as input, it outputs a correct sharing* $[ab]$*. The privacy is maintained when* $([r^{(0)}], \ldots, [r^{(3t+1)}])$ *are sharings of random values unknown to the adversary.* Multiplication *communicates* $\mathcal{O}(n^2\kappa)$ *bits.*

Lemma 35 *The protocol* ComputationPhase *(eventually) terminates for every honest player. Given that the* $\ell = (3t+1)c_M + c_R$ *sharings* $[r^{(1)}], \ldots, [r^{(\ell)}]$ *are correct t-sharings of random values unknown to the adversary, it computes the outputs of the circuit correctly and privately, while communicating* $\mathcal{O}(n^2 c_M \kappa + n c_O \kappa)$ *bits (where* c_M*,* c_R*, and* c_O *denote the number of multiplication, random, and output gates in the circuit, respectively).*

7.9 The Asynchronous MPC Protocol

The following protocol allows the players to evaluate an agreed arithmetic circuit C over a finite field $\mathbb{F}$: Denote the number of input, multiplication, random, and output gates as c_I, c_M, c_R, and c_O, respectively.

Protocol AsyncMPC **(C, c_I, c_M, c_R, c_O).**

1. Invoke PreparationPhase to generate $\ell = c_M(3t+1)+c_R$ random sharings.
2. Invoke InputPhase to let the players share their inputs.
3. Invoke ComputationPhase to evaluate the circuit (consisting of linear, multiplication, random, and output gates).

Theorem 6 *For every coalition of up to $t < n/4$ corrupted players and for every scheduler, the protocol* AsyncMPC *securely computes the circuit C.* AsyncMPC *communicates $\mathcal{O}\big((c_I n^2 + c_M n^3 + c_R n^2 + n c_O)\kappa + n^3\mathcal{BC}(\kappa)\big)$ bits and requires* 2 *invocations of* ACS,[47] *(which requires $\mathcal{O}(n^2\mathcal{BC}(\kappa))$).*

7.10 The Hybrid Model

7.10.1 Motivation

A big disadvantage of asynchronous networks is the fact that the inputs of up to t honest players cannot be considered in the computation. This restriction disqualifies fully asynchronous models for many real-world applications. Unfortunately, this drawback is intrinsic to the asynchronous model, no (what so ever clever) protocol can circumvent it. The only escape is to move to less general communication models, where at least some restriction on the scheduling of messages is given.

In [HNP05], an asynchronous (cryptographically secure) MPC protocol was presented in which all players can provide their inputs, given that one single round of communication is synchronous. However, this protocol has two serious drawbacks: First, the communication round which is required to be synchronous is round number 7 (we say that a message belongs to round k if it depends on a message received in round $k-1$). This essentially means that *the first* 7 *rounds* must be synchronous, because if not, then the synchronous round can never be started (the players would have to wait until all messages of round 6 are delivered — an endless wait in an asynchronous network).

[47]The protocol can easily be modified to use only a single invocation to ACS, by invoking PreparationPhase and InputPhase in parallel, and invoking ACS only once to find those dealers who have both correctly shared their input(s) as well as correctly shared enough random values.

The second drawback of this protocol is that one must a priori fix the mode in which the protocol is to be executed, namely either in the hybrid mode (with the risk that the protocol fails when some message in the first 7 rounds is not delivered synchronously), or in the fully asynchronous mode (with the risk that up to t honest players cannot provide their input, even when the network is synchronous).

7.10.2 Our Hybrid Model

We follow the approach of [HNP05], but strengthen it in both mentioned directions: First, we require only *the very first round* to be synchronous, and second, we guarantee that even if some messages in the first round are not delivered synchronously, still at least $n - t$ inputs are provided — so to speak the best of both worlds. A bit more precisely, we provide a fully asynchronous input protocol with the following properties:

- For every scheduler, the inputs of at least $n - t$ players are taken.
- If all messages sent by P_i at the very beginning of the computation are delivered within an a priory fixed time, then P_i's inputs are taken.

This means in particular that if the first round is fully synchronous, then the inputs of all honest players are taken, and if the network is fully asynchronous, then at least $n - t$ inputs are taken.

7.10.3 PrepareInputs and RestoreInput

We briefly describe the idea of the new input protocol (assuming, for the sake of simple notation, that every player gives exactly one input): In the first (supposedly synchronous) round, every player computes a degree-t Shamir-sharing of his input and sends one share to each player. Then, the players invoke the fully asynchronous input protocol, where the input of each player is a vector consisting of his real input, and his shares of the inputs of the other players. As result of the asynchronous input protocol, a core set C of at least $n - t$ players is found, whose input vectors are (eventually) t-shared among the players. For every player $P_i \in C$, the input is directly taken from his input vector. For every player $P_j \notin C$, the input is computed as follows: There are $n - t$ shares of his input, each t-shared as a component of the input vector of some player $P_i \in C$. Up to t of these players might be corrupted and have input a wrong share. Therefore, these t-shared shares are error-corrected and used as P_j's input. For error correction, $t+1$ random t-sharings are used. These will be generated (additionally) in the preparation phase. Then, right before the computation phase, sharings of the missing inputs are computed.

In the following, we present a (trivial) sub-protocol PrepareInputs, which prepares the inputs of all players (to be invoked in the first, supposedly synchronous round), and a protocol RestoreInput, which restores the sharing of an input $s^{(k)}$ of a player not in the core set, if possible (to be invoked right before the computation phase). The protocol RestoreInput needs $t+1$ t-sharings of random values, which must be generated in the preparation phase.

Protocol PrepareInputs **(every P_i holding input $s^{(i)}$).**

Code for player P_i:

1. Choose random degree-t polynomial $p(\cdot)$ with $p(0) = s^{(i)}$ and send to every P_j his share $s_j^{(i)} = p(\alpha_j)$.
2. Collect shares $s_i^{(j)}$ (from P_j) till the first round is over. Then compose your new input $\widetilde{s}^{(i)} = \left(s^{(i)}, s_i^{(1)}, \ldots, s_i^{(n)}\right)$, where $s_i^{(j)} = \perp$ if no share $s_i^{(j)}$ was received from P_j within the first round.

Protocol RestoreInput **(Core Set C, Input Sharings $[\widetilde{s}^{(i)}]$ of $P_i \in C$, $[r^{(0)}], \ldots, [r^{(t)}]$, $P_k \notin C$).**

Code for player P_i:

1. Define the blinding polynomial $b(x) = r^{(0)} + r^{(1)}x + \ldots + r^{(t)}x^t$, and for every P_j, define $[b_j] = [b(\alpha_j)] = [r^{(0)}] + [r^{(1)}]\alpha_j + \ldots + [r^{(t)}]\alpha_j^t$. Invoke Recons to reconstruct b_j towards P_j, for every P_j.
2. For every $P_j \in C$, denote by $[s_j^{(k)}]$ the sharing of P_j's share of P_k's input $s^{(k)}$. Note that $s_j^{(k)}$ is a part of the input vector $\widetilde{s}^{(j)}$. If $[s_j^{(k)}] \neq \perp$, then compute $[d_j] = [s_j^{(k)}] + [b_j]$, and invoke Recons to reconstruct d_j towards every player.
3. If there exists a degree-t polynomial $p(\cdot)$ such that at least $2t+1$ of the reconstructed values d_j lie on it, define $d'_i = p(\alpha_i)$, and compute your share $s_i^{(k)}$ of P_k's input $s^{(k)}$ as $d'_i - b_i$. The sharing of input $[s^{(k)}]$ was successfully restored. If no such polynomial $p(\cdot)$ exists, then $[s^{(k)}]$ cannot be restored.

Lemma 36 *The protocol* PrepareInputs *and* RestoreInput *terminate for all players. When all messages of a player P_k in Step 1 of* PrepareInputs *are synchronously delivered, then a sharing of his input $s^{(k)}$ can be successfully restored in* RestoreInput*, by any core set C with $C \geq n - t$ (with up to t cheaters). When an input sharing $[s^{(k)}]$ of an honest player P_k is restored in* RestoreInput*, then the shared value is the correct input of P_k. Furthermore, both* PrepareInputs *and* RestoreInput *preserve the privacy of inputs of honest players.*

Proof:[sketch] Termination and privacy are easy to verify. We focus on correctness. First assume that P_k is honest, and all his messages in Round 1 of PrepareInputs were synchronously delivered. Then every honest player P_i embeds the share $s_i^{(k)}$ in his input vector. There will be at least $n - t$ players in the core set, so at least $n - 2t$ honest players P_j. This means that there are at least $n - 2t$ t-consistent shares $s_j^{(k)}$, and hence, at least $n - 2t$ consistent shares d_j. For $t < n/4$, we have $n - 2t \geq 2t + 1$, and the result is a sharing of $d - b = (s^{(k)} + b) - b = s^{(k)}$. Then assume that P_k is honest, but not all his messages in Round 1 have been delivered synchronously. However, if there are $2t + 1$ points on the polynomial $p(\cdot)$, at least $t + 1$ of these points are from honest players, and hence the right input is restored.

7.10.4 The Hybrid MPC Protocol

The new main protocol for the hybrid model is as follows:

Protocol HybridMPC **(C, c_I, c_M, c_R, c_O).**

1. Invoke PrepareInputs to let every P_i with input $s^{(i)}$ share $s^{(i)}$ among all players.
2. Invoke PreparationPhase to generate $\ell = c_M(3t + 1) + c_R + c_I(t + 1)$ random sharings.
3. Invoke InputPhase (with P_i's input being the vector $\widetilde{s}^{(i)}$) to let the players share their input vectors.
4. Invoke RestoreInput to restore the inputs of every P_k not in the core set.
5. Invoke ComputationPhase to evaluate the circuit (consisting of linear, multiplication, random, and output gates).

Theorem 7 *For every coalition of up to $t < n/4$ corrupted players and for every scheduler, the protocol* HybridMPC *securely computes the circuit C, taking the inputs of all players (when the first round is synchronous), or taking the inputs of at least $n-t$ players (independently of any scheduling assumptions).* AsyncMPC *communicates $\mathcal{O}\big((c_I n^3 + c_M n^3 + c_R n^2 + n c_O)\kappa + n^3 \mathcal{BC}(\kappa)\big)$ bits and requires 2 invocations of* ACS *(can be reduced to 1).*

Chapter 8

Concluding Remarks

We have presented efficient MPC protocols for four different settings (each of them with optimal security parameters):

- We have shown that in the synchronous model perfectly secure MPC tolerating $t < n/3$ corrupted players is possible with the same communication complexity as passive, cryptographically and statistically secure MPC (with optimal thresholds).
- We have proposed a synchronous protocol statistically secure against an adversary corrupting a minority of the players, with communication complexity $\mathcal{O}(n^2\kappa)$ bits per multiplication. This protocol improves the complexity of the previous most efficient protocol for this setting which requires broadcasting $\mathcal{O}(n^5\kappa)$ bits per multiplication (and each of this broadcasts has to be simulated by an expensive broadcast protocol).
- Our synchronous Byzantine-agreement protocol with statistical security for $t < n/2$ communicates $\mathcal{O}(n^5\kappa)$ bits and thereby improves the previous most efficient BA protocol for this model by a factor of three.
- We have shown that in the asynchronous model, perfectly secure MPC for $t < n/4$ is possible communicating $\mathcal{O}(n^3\kappa)$ bits per multiplication. At the time of the publication, this was the complexity of the most efficient perfectly-secure protocol for the more restrictive synchronous setting.

We have introduced two new techniques which we believe to be of independent interest:

- Dispute control enables to limit the number of faults provoked by an active adversary by localizing a pair of players, one of them corrupted, every time a fault is detected and preventing this pair from disturbing the computation ever again.
- Hyper-invertible matrices enable to deterministically and very efficiently check the correctness of a bunch of sharings and extract a set of correct secret random sharings, given that a subset of the original sharings originate from honest players, and thus are correct, secret and random. Up to now, this was only possible with probabilistic correctness checks, resulting in protocols with unperfect security.

Bibliography

[BB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 201–210, August 1989.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proc. 25th ACM Symposium on the Theory of Computing (STOC)*, pages 52–61, 1993.

[Bea91a] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432, 1991.

[Bea91b] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, pages 75–122, 1991.

[BFKR90] Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. Security with low communication overhead. In *Advances in Cryptology — CRYPTO '90*, volume 537 of *Lecture Notes in Computer Science*, pages 62–76, 1990.

[BGP92] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Bit optimal distributed consensus. *Computer Science Research*, pages 313–322, 1992. Preliminary version in STOC'89.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th ACM Symposium on the Theory of Computing (STOC)*, pages 1–10, 1988.

[BH06] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography — TCC 2006*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328. Springer-Verlag, March 2006.

[BH07] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous MPC. In Kaoru Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 376–392. Springer-Verlag, December 2007.

[BH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *Theory of Cryptography — TCC 2008*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer-Verlag, March 2008.

[BHR07] Zuzana Beerliová-Trubíniová, Martin Hirt, and Micha Riser. Efficient Byzantine agreement with faulty minority. In Kaoru Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 393 – 409. Springer-Verlag, December 2007.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, 1994.

[Bla79] George Robert Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference 1979*, volume 48 of *American Federation of Information Processing Societies Proceedings*, pages 313–317, 1979.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proc. 22nd ACM Symposium on the Theory of Computing (STOC)*, pages 503–513, 1990.

[BPW91] Birgit Baum-Waidner, Birgit Pfitzmann, and Michael Waidner. Unconditional byzantine agreement with good majority. In *8th Annual Symposium on Theoretical Aspects of Computer Science*, volume 480 of *lncs*, pages 285–295, Hamburg, Germany, February 1991. Springer.

[BPW04] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A general composition theorem for secure reactive systems. In Moni Naor, editor, *Theory of Cryptography — TCC 2004*, volume 2951 of *Lecture Notes in Computer Science*, pages 336–354. Springer-Verlag, 2004.

[Bra84] Gabriel Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[Can95] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, Rehovot 76100, Israel, June 1995.

[Can98] Ran Canetti. Security and composition of multi-party cryptographic protocols. Manuscript, June 1998. Former (more general) version: Modular composition of multi-party cryptographic protocols, Nov. 1997.

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42st IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 15–17, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proc. 20th ACM Symposium on the Theory of Computing (STOC)*, pages 11–19, 1988.

[CDD$^+$99] Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 311–326, 1999.

[CDF01] Ronald Cramer, Ivan Damgård, and Serge Fehr. On the cost of reconstructing a secret, or VSS with optimal reconstruction phase. In Joe Kilian, editor, *Advances in Cryptology —*

CRYPTO 2001, volume 2139 of *Lecture Notes in Computer Science*, pages 503–523. Springer-Verlag, 2001.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In *Advances in Cryptology — CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 87–119. Springer-Verlag, 1987.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, 2001.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences (JCSS)*, 18(4):143–154, 1979. Preliminary version has appeared in Proc. 9st STOC, 1977.

[CW92] Brian A. Coan and Jennifer L. Welch. Modular construction of a Byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, March 1992. Preliminary version in PODC'89.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology — CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer-Verlag, 2005.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 501–520. Springer-Verlag, 2006.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Alfred Menezes, editor, *Advances in Cryptology — CRYPTO 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983. Preliminary version has appeared in Proc. 14th STOC, 1982.

[Fit03] Matthias Fitzi. *Generalized Communication and Security Models in Byzantine Agreement*. PhD thesis, ETH Zurich, 2003. Reprint as vol. 4 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-853-3, Hartung-Gorre Verlag, Konstanz, 2003.

[Fit04] Matthias Fitzi. Personal communication, 2004.

[FLM86] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *ACM Symposium on the Theory of Computing (STOC) '92*, pages 699–710. ACM, 1992.

[GHY87] Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure fault-tolerant protocols and the public-key model. In *Advances in Cryptology — CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 135–155. Springer-Verlag, 1987.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proc. 19th ACM Symposium on the Theory of Computing (STOC)*, pages 218–229, 1987.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 101–111, 1998.

[HM01] Martin Hirt and Ueli Maurer. Robustness for free in unconditional multi-party computation. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 101–118. Springer-Verlag, August 2001.

[HMP00] Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 143–161. Springer-Verlag, December 2000.

[HN05] Martin Hirt and Jesper Buus Nielsen. Upper bounds on the communication complexity of optimally resilient cryptographic multiparty computation. In Bimal Roy, editor, *Advances in Cryptology — ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 79–99. Springer-Verlag, December 2005.

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *Advances in Cryptology — CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer-Verlag, August 2006.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. In Ronald Cramer, editor, *Advances in Cryptology — EUROCRYPT2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 322–340. Springer-Verlag, May 2005.

[HNP08] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. In Luca Aceto, Magnus M. Halldorsson, and Anna Ingolfsdottir, editors, *Automata, Languages and Programming — ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 473–485. Springer-Verlag, July 2008.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MR98] Silvio Micali and Phillip Rogaway. Secure computation: The information theoretic case. Manuscript, 1998. Former version: Secure computation, In *Advances in Cryptology — CRYPTO '91*, volume 576 of *Lecture Note in Computer Science*, pp. 392–404, Springer-Verlag, 1991.

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

[PSR02] B. Prabhu, K. Srinathan, and C. Pandu Rangan. Asynchronous unconditionally secure computation: An efficiency improvement. In *Proc. Indocrypt 2002*, Lecture Notes in Computer Science, 2002.

[PW92] Birgit Pfitzmann and Michael Waidner. Unconditional Byzantine agreement for any number of faulty processors. In *Proc. 9th Symposium on Theoretical Aspects of Computer Science — STACS '92*, volume 577 of *Lecture Notes in Computer Science*, 1992.

[PW96] Birgit Pfitzmann and Michael Waidner. Information-theoretic pseudosignatures and byzantine agreement for $t >= n/3$. Technical report, IBM Research, 1996.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proc. 21st ACM Symposium on the Theory of Computing (STOC)*, pages 73–85, 1989.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[SHZI02] Junji Shikata, Goichiro Hanaoka, Yuliang Zheng, and Hideki Imai. Security notions for unconditionally secure signature schemes. In *Advances in Cryptology — EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 434–449, 2002.

[SR00] K. Srinathan and C. Pandu Rangan. Efficient asynchronous secure multiparty distributed computation. In *Proc. Indocrypt 2000*, Lecture Notes in Computer Science, December 2000.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 160–164. IEEE, 1982.

Personal Details

Name	Beerliová-Trubíniová Zuzana
Address	Berninastr. 104, CH-8057 Zrich
Phone	+41 44 340 02 83
E-Mail	bzuzana@inf.ethz.ch
Born	26.06.1977 in Bratislava (Slovakia)
Languages	Slovak, German, English

Education

1995	Matura at Gymnasium Jura Hronca in Bratislava
1995 – 1998	studies at University of Economics Bratislava, SK
1998 – 2004	M.Sc. ETH in Mathematics, ETH Zurich
2004 – 2008	Ph. D. student at ETH Zurich, Information Security and Cryptography Research Group

Publications

- Z. Beerliová-Trubíniová, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, M. Mihalák, and L.S. Ram: *Network discovery and verification*. Selected Areas in Communications 24(12), IEEE, 2006.
- Z. Beerliová-Trubíniová and M. Hirt: *Efficient Multi-Party Computation with Dispute Control*, TCC 2006, LNCS 3876, Springer-Verlag, 2006.
- Z. Beerliová-Trubíniová and M. Hirt: *Simple and efficient perfectly-secure asynchronous MPC*. ASIACRYPT 2007, LNCS 4833, Springer-Verlag, 2007.
- Z. Beerliová-Trubíniová, M. Hirt, and M. Riser: *Efficient Byzantine agreement with faulty minority*, ASIACRYPT 2007, LNCS 4833, Springer-Verlag, 2007.
- Z. Beerliová-Trubíniová, M. Fitzi, M. Hirt, U. Maurer, and V. Zikas: *Secure Multi-Party Computation vs. Secure Function Evaluation*, TCC 2008, LNCS 4948, Springer-Verlag, 2008.
- Z. Beerliová-Trubíniová and M. Hirt: *Perfectly-secure MPC with linear communication complexity*, TCC 2008, LNCS 4948, Springer-Verlag, 2008.