J Cryptol (2017) 30:58–151
DOI: 10.1007/s00145-015-9214-4

Journal of CRYPTOLOGY

A Full Proof of the BGW Protocol for Perfectly Secure Multiparty Computation*

Gilad Asharov†
School of Computer Science and Engineering, Hebrew University of Jerusalem, Jerusalem, Israel
asharog@cs.biu.ac.il

Yehuda Lindell
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
Yehuda.Lindell@biu.ac.il

Communicated by Goldreich.

Received 8 July 2011
Online publication 28 September 2015

Abstract. In the setting of secure multiparty computation, a set of n parties with private inputs wish to jointly compute some functionality of their inputs. One of the most fundamental results of secure computation was presented by Ben-Or, Goldwasser, and Wigderson (BGW) in 1988. They demonstrated that any n-party functionality can be computed with *perfect security*, in the private channels model. When the adversary is semi-honest, this holds as long as $t < n/2$ parties are corrupted, and when the adversary is malicious, this holds as long as $t < n/3$ parties are corrupted. Unfortunately, a full proof of these results was never published. In this paper, we remedy this situation and provide a full proof of security of the BGW protocol. This includes a full description of the protocol for the malicious setting, including the construction of a new subprotocol for the perfect multiplication protocol that seems necessary for the case of $n/4 \leq t < n/3$.

Keywords. Multiparty computation, Perfect security, BGW, Cryptographic protocols.

1. Introduction

1.1. *Background: Secure Computation*

In the setting of secure multiparty computation, a set of n parties with possibly private inputs wish to securely compute some function of their inputs in the presence of adver-

* This work was funded by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement No. 239868, and by the THE ISRAEL SCIENCE FOUNDATION (Grant No. 189/11).

† The work was done while the author was a Ph.D. student at Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel.

sarial behavior. Loosely speaking, the security requirements from such a computation are that nothing is learned from the protocol other than the output (*privacy*), that the output is distributed according to the prescribed functionality (*correctness*), that parties cannot choose their inputs as a function of the others' inputs (*independence of inputs*), and that all parties receive output (*fairness* and *guaranteed output delivery*). The actual definition [4,8,20,22,29] formalizes this by comparing the result of a real protocol execution with the result of an ideal execution in an ideal model where an incorruptible trusted party carries out the computation for the parties. This definition has come to be known as the "ideal/real simulation paradigm."

There are many different settings within which secure computation has been considered. Regarding the adversary, one can consider semi-honest adversaries (who follow the protocol specification but try to learn more than they should by inspecting the protocol transcript) or malicious adversaries (who may follow an arbitrary strategy). In addition, an adversary may be limited to polynomial time (as in the computational setting) or unbounded (as in the information-theoretic setting). Finally, the adversary may be static (meaning that the set of corrupted parties is fixed before the protocol execution begins) or adaptive (meaning that the adversary can adaptively choose to corrupt throughout the protocol execution).

Wide reaching feasibility results regarding secure multiparty computation were presented in the mid- to late 1980s. The first feasibility results for secure computation were in the computational setting and were provided by Yao [34] for the two-party case, and by Goldreich et al. [21] for the multiparty case. These results begged the question as to whether it is possible to avoid computational hardness assumptions, that is, provide analogous results for the information-theoretic setting. This question was answered in the affirmative by Ben-Or et al. [7], Chaum et al. [14] who showed that when less than a third of the parties are corrupted, it is possible to securely compute any functionality in the information-theoretic setting, assuming an ideal private channel between each pair of parties. The protocol of Ben-Or et al. [7] achieved *perfect* security, while the protocol of Chaum et al. [14] achieved *statistical* security. These results were followed by Rabin and Ben-Or [31], Beaver [3] who showed that if the parties are also given an ideal broadcast channel, then it is possible to securely compute any functionality with statistical security assuming only an honest majority.

1.2. *The BGW Protocol*

Our focus is on the results of Ben-Or, Goldwasser, and Wigderson (BGW) [7], who showed that every functionality can be computed with *perfect security* in the presence of semi-honest adversaries controlling a minority of parties, and in the presence of malicious adversaries controlling less than a third of the parties. The discovery that secure computation can be carried out information theoretically, and the techniques used by BGW, was highly influential. In addition, as we shall see, the fact that security is *perfect*—informally meaning that there is a *zero probability* of cheating by the adversary—provides real security advantages over protocols that have a negligible probability of failure (cf. [24]). For this reason, we focus on the BGW protocol [7] rather than on [14].

On a high level, the BGW protocol works by having the parties compute the desired function f (from n inputs to n outputs) by securely emulating the computation of an arithmetic circuit computing f. In this computation, the parties compute shares of the output of a circuit gate given shares of the input wires of that gate. To be more exact, the parties first share their inputs with each other using Shamir's secret sharing [32]; in the case of malicious adversaries, a *verifiable* secret-sharing protocol (cf. [15,21]) is used. The parties then emulate the computation of each gate of the circuit, computing Shamir-shares of the gate's output from the Shamir-shares of the gate's inputs. As we shall see, this secret sharing has the property that addition gates in the circuit can be emulated using local computation only. Thus, the parties only interact in order to emulate the computation of multiplication gates; this step is the most involved part of the protocol. Finally, the parties reconstruct the secrets from the shares of the output wires of the circuit in order to obtain their output.

We proceed to describe the protocol in a bit more detail. Shamir's secret sharing enables the sharing of a secret s among n parties, so that any subset of $t+1$ or more parties can efficiently reconstruct the secret, and any subset of t or less parties learn no information whatsoever about the secret. Let $\mathbb{F}$ be a finite field of size greater than n, let $\alpha_1, \dots, \alpha_n$ be n distinct nonzero field elements, and let $s \in \mathbb{F}$. Then, in order to share s, a polynomial $p(x) \in \mathbb{F}[x]$ of degree t with constant term s is randomly chosen, and the share of the ith party P_i is set to $p(\alpha_i)$. By interpolation, given any $t+1$ points, it is possible to reconstruct p and compute $s = p(0)$. Furthermore, since p is random, its values at any t or less of the α_i's give no information about s.

Now, let n denote the number of parties participating in the multiparty computation, and let t be a bound on the number of corrupted parties. The first step of the BGW protocol is for all parties to share their inputs using Shamir's secret-sharing scheme. In the case of semi-honest adversaries, plain Shamir sharing with a threshold $t < n/2$ is used, and in the case of malicious adversaries verifiable secret sharing (VSS) with a threshold $t < n/3$ is used. A verifiable secret-sharing protocol is needed for the case of malicious adversaries in order to prevent cheating, and the BGW paper was also the first to construct a *perfect* VSS protocol.

Next, the parties emulate the computation of the gates of the circuit. The first observation is that addition gates can be computed locally. That is, given shares $p(\alpha_i)$ and $q(\alpha_i)$ of the two input wires to an addition gate, it holds that $r(\alpha_i) = p(\alpha_i) + q(\alpha_i)$ is a valid sharing of the output wire. This is due to the fact that the polynomial $r(x)$ defined by the sum of the shares has the same degree as both $p(x)$ and $q(x)$, and $r(0) = p(0) + q(0)$.

Regarding multiplication gates, observe that by computing $r(\alpha_i) = p(\alpha_i) \cdot q(\alpha_i)$, the parties obtain shares of a polynomial $r(x)$ with constant term $p(0) \cdot q(0)$ as desired. However, the degree of $r(x)$ is $2t$, since the degrees of $p(x)$ and $q(x)$ are both t. Since reconstruction works as long as the polynomial used for the sharing is of degree t, this causes a problem. Thus, the multiplication protocol works by *reducing the degree* of the polynomial $r(x)$ back to t. In the case of semi-honest parties, the degree reduction can be carried out as long as $t < n/2$ (it is required that $t < n/2$ since otherwise the degree of $r(x) = p(x) \cdot q(x)$ will be greater than or equal to n, which is not fully defined by the n parties' shares). In the case of malicious parties, the degree reduction is much more complex and works as long as $t < n/3$. In order to obtain some intuition as to why $t < n/3$ is needed, observe that Shamir's secret sharing can also be viewed as a Reed–Solomon code of the polynomial [28]. With a polynomial of degree t, it is possible to correct up $(n-t-1)/2$ errors. Setting $t < n/3$, we have that $n \geq 3t+1$, and so

$(n - t - 1)/2 \geq t$ errors can be corrected. This means that if up to t malicious parties send incorrect values, the honest parties can use error correction and recover. Indeed, the BGW protocol in the case of malicious adversaries relies heavily on the use of error correction in order to prevent the adversary from cheating.

We remark that $t < n/3$ is not merely a limitation of the way the BGW protocol works. In particular, the fact that at most $t < n/3$ corruptions can be tolerated in the malicious model follows immediately from the fact that at most $t < n/3$ corruptions can be tolerated for Byzantine agreement [30]. In contrast, given a broadcast channel, it *is* possible to securely compute any functionality with information-theoretic (statistical) security for any $t < n/2$ [3,31].

1.3. *Our Results*

Despite the importance of the BGW result, a full proof of its security has never appeared (and this is also the state of affairs regarding [14]). In addition, a full description of the protocol in the malicious setting was also never published. In this paper, we remedy this situation and provide a full description and proof of the BGW protocols, for both the semi-honest and malicious settings. We prove security relative to the ideal/real definition of security for multiparty computation. This also involves carefully defining the functionalities and subfunctionalities that are used in order to achieve the result, as needed for presenting a modular proof. Our main result is a proof of the following informally stated theorem:

Theorem 1. (basic security of the BGW protocol—informally stated) *Consider a synchronous network with pairwise private channels and a broadcast channel. Then:*

1. *Semi-honest: For every n-ary functionality f, there exists a protocol for computing f with perfect security in the presence of a static semi-honest adversary controlling up to $t < n/2$ parties;*
2. *Malicious: For every n-ary functionality f, there exists a protocol for computing f with perfect security in the presence of a static malicious adversary controlling up to $t < n/3$ parties.*

The communication complexity of the protocol is $O(\mathsf{poly}(n) \cdot |C|)$ where C is an arithmetic circuit computing f, and the round complexity is linear in the depth of the circuit C.

All of our protocols are presented in a model with pairwise private channels *and* secure broadcast. Since we only consider the case of $t < n/3$ malicious corruptions, secure broadcast can be achieved in a synchronous network with pairwise channels by running Byzantine Generals [18,25,30]. In order to obtain (expected) round complexity linear in the depth of $|C|$, an expected constant-round Byzantine Generals protocol of Feldman and Micali [18] (with composition as in [6,27]) is used.

Security Under Composition Theorem 1 is proven in the classic setting of a static adversary and stand-alone computation, where the latter means that security is proven for the case that only a single protocol execution takes place at a time. Fortunately, it was shown in [24] that any protocol that is *perfectly* secure and has a black-box nonrewinding simulator is also secure under universal composability [9] (meaning that security is guaranteed to hold when many arbitrary protocols are run concurrently with the secure protocol). Since our proof of security satisfies this condition, we obtain the following corollary, which relates to a far more powerful adversarial setting:

Corollary 2. (UC information-theoretic security of the BGW protocol) *Consider a synchronous network with private channels. Then, for every n-ary functionality f, there exists a protocol for computing f with perfect* universally composable security *in the presence of an static semi-honest adversary controlling up to $t < n/2$ parties, and there exists a protocol for computing f with perfect* universally composable security *in the presence of a static malicious adversary controlling up to $t < n/3$ parties.*

Corollary 2 refers to information-theoretic security in the ideal private channels model. We now derive a corollary to the computational model with authenticated channels only. In order to derive this corollary, we first observe that information-theoretic security implies security in the presence of polynomial-time adversaries (this holds as long as the simulator is required to run in time that is polynomial in the running time of the adversary, as advocated in [20, Sec. 7.6.1]). Furthermore, the ideal private channels of the information-theoretic setting can be replaced with computationally secure channels that can be constructed over authenticated channels using semantically secure public-key encryption [23,33]. We have:

Corollary 3. (UC computational security of the BGW protocol) *Consider a synchronous network with authenticated channels. Assuming the existence of semantically secure public-key encryption, for every n-ary functionality f, there exists a protocol for computing f with* universally composable security *in the presence of a static malicious adversary controlling up to $t < n/3$ parties.*

We stress that unlike the UC-secure computational protocols of Canetti et al. [13] (that are secure for any $t < n$), the protocols of Corollary 3 are in the *plain model*, with authenticated channels but with no other trusted setup (in particular, no common reference string). Although well-accepted folklore, Corollaries 2 and 3 have never been proved. Thus, our work also constitutes the first full proof that universally composable protocols exist in the plain model (with authenticated channels) for any functionality, in the presence of static malicious adversaries controlling any $t < n/3$ parties.

Adaptive Security with Inefficient Simulation We also conclude security for the case of adaptive corruptions (see, [8,11]). In [10] it was shown that any protocol that is proven perfectly secure under the security definition of Dodis and Micali [16] is also secure in the presence of adaptive adversaries, alas with inefficient simulation. We use this to derive security in the presence of adaptive adversaries, albeit with the weaker guarantee provided by inefficient simulation (in particular, this does not imply adaptive security in the computational setting). See Sect. 8 for more details.[1]

[1] In previous versions of this paper [1] and in [2], we mistakenly stated that using [10] it is possible to obtain full adaptive security with efficient simulation. However, this is actually not known, and [10] only proves that perfect security under [16] implies adaptive security with *inefficient* simulation, which is significantly weaker.

Organization In Sect. 2, we present a brief overview of the standard definitions of perfectly secure multiparty computation and of the modular sequential composition theorem that is used throughout in our proofs. Then, in Sect. 3, we describe Shamir's secret-sharing scheme and rigorously prove a number of useful properties of this scheme. In Sect. 4 we present the BGW protocol for the case of semi-honest adversaries. An overview of the overall construction appears in Sect. 4.1, and an overview of the multiplication protocol appears at the beginning of Sect. 4.3.

The BGW protocol for the case of malicious adversaries is presented in Sects. 5–7. In Sect. 5 we present the BGW verifiable secret-sharing (VSS) protocol that uses bivariate polynomials. This section includes background on Reed–Solomon encoding and properties of bivariate polynomials that are needed for proving the security of the VSS protocol. Next, in Sect. 6, we present the most involved part of the protocol—the multiplication protocol for computing shares of the product of shares. This involves a number of steps and subprotocols, some of which are new. The main tool for the BGW multiplication protocol is a subprotocol for verifiably sharing the product of a party's shares. This subprotocol, along with a detailed discussion and overview, is presented in Sect. 6.6. Our aim has been to prove the security of the original BGW protocol. However, where necessary, some changes were made to the multiplication protocol as described originally in [7]. Finally, in Sect. 7, the final protocol for secure multiparty computation is presented. The protocol is proven secure for any VSS and multiplication protocols that securely realize the VSS and multiplication functionalities that we define in Sects. 5 and 6, respectively. In addition, an exact count of the communication complexity of the BGW protocol for malicious adversaries is given. We conclude in Sect. 8 by showing how to derive security in other settings (adaptive adversaries, composition, and the computational setting).

2. Preliminaries and Definitions

In this section, we review the definition of perfect security in the presence of semi-honest and malicious adversaries. We refer the reader to [20, Sec. 7.6.1] and [8] for more details and discussion.

In the definitions below, we consider the *stand-alone* setting with a *synchronous* network, and perfectly *private channels* between all parties. For simplicity, we will also assume that the parties have a broadcast channel; as is standard, this can be implemented using an appropriate Byzantine generals protocol [25,30]. Since we consider synchronous channels and the computation takes place in clearly defined rounds, if a message is not received in a given round, then this fact is immediately known to the party who is supposed to receive the message. Thus, we can write "if a message is not received" or "if the adversary does not send a message" and this is well defined. We consider *static corruptions* meaning that the set of corrupted parties is fixed ahead of time, and the *stand-alone setting* meaning that only a single protocol execution takes place; extensions to the case of adaptive corruptions and composition are considered in Sect. 8.

Basic Notation For a set A, we write $a \in_R A$ when a is chosen uniformly from A. We denote the number of parties by n, and a bound on the number of corrupted parties by t. Let $f\colon \left(\{0,1\}^*\right)^n \to \left(\{0,1\}^*\right)^n$ be a possibly probabilistic n-ary functionality, where

$f_i(x_1,\ldots,x_n)$ denotes the ith element of $f(x_1,\ldots,x_n)$. We denote by $I=\{i_1,\ldots i_\ell\}\subset[n]$ the indices of the corrupted parties, where $[n]$ denotes the set $\{1,\ldots,n\}$. By the above, $|I|\le t$. Let $\vec{x}=(x_1,\ldots,x_n)$, and let $\vec{x}_I$ and $f_I(\vec{x})$ denote projections of the corresponding n-ary sequence on the coordinates in I; that is, $\vec{x}_I=(x_{i_1},\ldots,x_{i_\ell})$ and $f_I(\vec{x})=(f_{i_1}(\vec{x}),\ldots,f_{i_\ell}(\vec{x}))$. Finally, to ease the notation, we omit the index i when we write the set $\{(i,a_i)\}_{i=1}^n$ and simply write $\{a_i\}_{i=1}^n$. Thus, for instance, the set of shares $\{(i_1,f(\alpha_{i_1})),\ldots,(i_\ell,f(\alpha_{i_\ell}))\}$ is denoted as $\{f(\alpha_i)\}_{i\in I}$.

Terminology In this paper, we consider security in the presence of both semi-honest and malicious adversaries. As in [20], we call security in the presence of a semi-honest adversary controlling t parties t-privacy, and security in the presence of a malicious adversary controlling t parties t-security. Since we only deal with perfect security in this paper, we use the terms t-private and t-secure without any additional adjective, with the understanding that the privacy/security is always perfect.

2.1. *Perfect Security in the Presence of Semi-honest Adversaries*

We are now ready to define security in the presence of semi-honest adversaries. Loosely speaking, the definition states that a protocol is t-private if the view of up to t corrupted parties in a real protocol execution can be generated by a simulator given only the corrupted parties' inputs and outputs.

The view of the ith party P_i during an execution of a protocol π on inputs $\vec{x}$, denoted $\textsc{view}_i^\pi(\vec{x})$, is defined to be $(x_i,r_i;m_{i_1},\ldots,m_{i_k})$ where x_i is P_i's private input, r_i is its internal coin tosses, and m_{i_j} is the jth message that was received by P_i in the protocol execution. For every $I=\{i_1,\ldots i_\ell\}$, we denote $\textsc{view}_I^\pi(\vec{x})=(\textsc{view}_{i_1}^\pi(\vec{x}),\ldots\textsc{view}_{i_\ell}^\pi(\vec{x}))$. The output of all parties from an execution of π on inputs $\vec{x}$ is denoted $\textsc{output}^\pi(\vec{x})$; observe that the output of each party can be computed from its own (private) view of the execution.

We first present the definition for deterministic functionalities, since this is simpler than the general case of probabilistic functionalities.

Definition 2.1. (*t-privacy of n-party protocols—deterministic functionalities*) Let $f\colon\left(\{0,1\}^*\right)^n\to\left(\{0,1\}^*\right)^n$ be a *deterministic* n-ary functionality and let π be a protocol. We say that π is t-private for f if for every $\vec{x}\in(\{0,1\}^*)^n$ where $|x_1|=\cdots=|x_n|$,

$$\textsc{output}^\pi(x_1,\ldots,x_n)=f(x_1,\ldots,x_n) \tag{2.1}$$

and there exists a probabilistic polynomial-time algorithm $\mathcal{S}$ such that for every $I\subset[n]$ of cardinality at most t, and every $\vec{x}\in(\{0,1\}^*)^n$ where $|x_1|=\cdots=|x_n|$, it holds that:

$$\left\{\mathcal{S}\left(I,\vec{x}_I,f_I\left(\vec{x}\right)\right)\right\}\equiv\left\{\textsc{view}_I^\pi\left(\vec{x}\right)\right\} \tag{2.2}$$

The above definition separately considers the issue of output correctness (Eq. 2.1) and privacy (Eq. 2.2), where the latter captures privacy since the ability to generate the corrupted parties' view given only the input and output means that nothing more than the input and output is learned from the protocol execution. However, in the case of probabilistic functionalities, it is necessary to intertwine the requirements of privacy and correctness and consider the *joint* distribution of the output of $\mathcal{S}$ and of the parties; see [8,20] for discussion. Thus, in the general case of probabilistic functionalities, the following definition of t-privacy is used.

Definition 2.2. (*t-privacy of n-party protocols—general case*) Let $f\colon \left(\{0,1\}^*\right)^n \to \left(\{0,1\}^*\right)^n$ be a probabilistic n-ary functionality and let π be a protocol. We say that π is t-private for f if there exists a probabilistic polynomial-time algorithm $\mathcal{S}$ such that for every $I \subset [n]$ of cardinality at most t, and every $\vec{x} \in (\{0,1\}^*)^n$ where $|x_1| = \cdots = |x_n|$, it holds that:

$$\left\{(\mathcal{S}(I, \vec{x}_I, f_I(\vec{x})), f(\vec{x}))\right\} \equiv \left\{(\textsc{view}_I^{\pi}(\vec{x}), \textsc{output}^{\pi}(\vec{x}))\right\}. \tag{2.3}$$

We remark that in the case of deterministic functionalities, the separate requirements of Eqs. (2.1) and (2.2) actually imply the joint distribution of Eq. (2.3). This is due to the fact that when f is deterministic, $f(\vec{x})$ is a single value and not a distribution.

Our Presentation—Deterministic Functionalities For the sake of simplicity and clarity, we present the BGW protocol and prove its security for the case of deterministic functionalities only. This enables us to prove the overall BGW protocol using Definition 2.1, which makes the proof significantly simpler. Fortunately, this does not limit our result since it has already been shown that it is possible to t-privately compute *any probabilistic functionality* using a general protocol for t-privately computing any deterministic functionality; see [20, Sec. 7.3.1].

2.2. *Perfect Security in the Presence of Malicious Adversaries*

We now consider malicious adversaries that can follow an arbitrary strategy in order to carry out their attack; we stress that the adversary is not required to be efficient in any way. Security is formalized by comparing a real protocol execution to an ideal model where the parties just send their inputs to the trusted party and receive back outputs. See [8,20] for details on how to define these real and ideal executions; we briefly describe them here.

Real Model In the real model, the parties run the protocol π. We consider a synchronous network with private point-to-point channels, and an authenticated broadcast channel. This means that the computation proceeds in rounds, and in each round parties can send private messages to other parties and can broadcast a message to all other parties. We stress that the adversary cannot read or modify messages sent over the point-to-point channels, and that the broadcast channel is authenticated, meaning that all parties know who sent the message and the adversary cannot tamper with it in any way. Nevertheless, the adversary is assumed to be *rushing*, meaning that in every given round it can see the messages sent by the honest parties before it determines the messages sent by the corrupted parties.

Let π be a n-party protocol, let $\mathcal{A}$ be an arbitrary machine with auxiliary input z, and let $I \subset [n]$ be the set of corrupted parties controlled by $\mathcal{A}$. We denote by $\textsc{real}_{\pi,\mathcal{A}(z),I}(\vec{x})$ the random variable consisting of the view of the adversary $\mathcal{A}$ and the outputs of the honest parties, following a real execution of π in the aforementioned real model, where for every $i \in [n]$, party P_i has input x_i.

Ideal Model In the ideal model for a functionality f, the parties send their inputs to an incorruptible trusted party who computes the output for them. We denote the ideal adversary by $\mathcal{S}$ (since it is a "simulator") and the set of corrupted parties by I. An execution in the ideal model works as follows:

- **Input stage**: The adversary $\mathcal{S}$ for the ideal model receives auxiliary input z and sees the inputs x_i of the corrupted parties P_i (for all $i \in I$). $\mathcal{S}$ can substitute any x_i with any x'_i of its choice under the condition that $|x'_i| = |x_i|$.
- **Computation**: Each party sends its (possibly modified) input to the trusted party; it denotes the inputs sent by $x'_1, \ldots, x'_n$. The trusted party computes $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$ and sends y_j to P_j, for every $j \in [n]$.
- **Outputs**: Each honest party P_j $(j \notin I)$ outputs y_j, the corrupted parties output $\perp$, and the adversary $\mathcal{S}$ outputs an arbitrary function of its view.

Throughout the paper, we will refer to communication between the parties and the functionality. For example, we will often write that a party sends its input to the functionality; this is just shorthand for saying that the input is sent to the trusted party who computes the functionality.

We denote by $\text{IDEAL}_{f,\mathcal{S}(z),I}(\vec{x})$ the outputs of the ideal adversary $\mathcal{S}$ controlling the corrupted parties in I and of the honest parties after an ideal execution with a trusted party computing f, upon inputs $x_1, \ldots, x_n$ for the parties and auxiliary input z for $\mathcal{S}$. We stress that the communication between the trusted party and $P_1, \ldots, P_n$ is over an ideal private channel.

Definition of Security Informally, we say that a protocol is secure if its real-world behavior can be emulated in the ideal model. That is, we require that for every real-model adversary $\mathcal{A}$ there exists an ideal model adversary $\mathcal{S}$ such that the result of a real execution of the protocol with $\mathcal{A}$ has the same distribution as the result of an ideal execution with $\mathcal{S}$. This means that the adversarial capabilities of $\mathcal{A}$ in a real protocol execution are just what $\mathcal{S}$ can do in the ideal model.

In the definition of security, we require that the ideal model adversary $\mathcal{S}$ run in time that is polynomial in the running time of $\mathcal{A}$, whatever the latter may be. As argued in [8,20] this definitional choice is important since it guarantees that information-theoretic security implies computational security. In such a case, we say that $\mathcal{S}$ is of comparable complexity to $\mathcal{A}$.

Definition 2.3. Let $f\colon \left(\{0,1\}^*\right)^n \to \left(\{0,1\}^*\right)^n$ be an n-ary functionality and let π be a protocol. We say that π is t-secure for f if for every probabilistic adversary $\mathcal{A}$ in the real model, there exists a probabilistic adversary $\mathcal{S}$ of comparable complexity in the ideal model, such that for every $I \subset [n]$ of cardinality at most t, every $\vec{x} \in (\{0,1\}^*)^n$ where $|x_1| = \cdots = |x_n|$, and every $z \in \{0,1\}^*$, it holds that:

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),I}(\vec{x})\right\} \equiv \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x})\right\}.$$

Reactive Functionalities The above definition refers to functionalities that map inputs to outputs in a single computation. However, some computations take place in stages, and state is preserved between stages. Two examples of such functionalities are mental poker (where cards are dealt and thrown and redealt [21]) and commitment schemes (where there is a separate commitment and decommitment phase; see [9] for a definition of commitments via an ideal functionality). Such functionalities are called reactive, and the definition of security is extended to this case in the straightforward way by allowing the trusted party to obtain inputs and send outputs in phases; see [20, Section 7.7.1.3].

2.3. *Modular Composition*

The sequential modular composition theorem [8] is an important tool for analyzing the security of a protocol in a modular way. Let π_f be a protocol for securely computing f that uses a subprotocol π_g for computing g. Then, the theorem states that it suffices to consider the execution of π_f in a hybrid model where a trusted third party is used to ideally compute g (instead of the parties running the real subprotocol π_g). This theorem facilitates a modular analysis of security via the following methodology: First prove the security of π_g, and then prove the security of π_f in a model allowing an ideal party for g. The model in which π_f is analyzed using ideal calls to g, instead of executing π_g, is called the g-hybrid model because it involves both a real protocol execution and an ideal trusted third party computing g.

More formally, in the hybrid model, the parties all have oracle tapes for some oracle (trusted party) that computes the functionality g. Then, if the real protocol π_f instructs the parties to run the subprotocol π_g using inputs $u_1, \ldots, u_n$, then each party P_i simply writes u_i to its outgoing oracle tape. Then, in the next round, it receives back the output $g_i(u_1, \ldots, u_n)$ on its incoming oracle tape. We denote by $\textsc{hybrid}^{g}_{\pi_f,\mathcal{A}(z),I}(\vec{x})$ an execution of protocol π_f where each call to π_g is carried out using an oracle computing g. See [8,20] for a formal definition of this model for both the semi-honest and malicious cases, and for proofs that if π_f is t-private (resp., t-secure) for f in the g-hybrid model, and π_g is t-private (resp., t-secure) for g, then π_f when run in the real model using π_g is t-private (resp., t-secure) for f.

3. Shamir's Secret-Sharing Scheme [32] and Its Properties

3.1. *The Basic Scheme*

A central tool in the BGW protocol is Shamir's secret-sharing scheme [32]. Roughly speaking, a $(t+1)$-out-of-n secret-sharing scheme takes as input a secret s from some domain, and outputs n shares, with the property that it is possible to efficiently reconstruct s from every subset of $t+1$ shares, but every subset of t or less shares reveals nothing about the secret s. The value $t+1$ is called the threshold of the scheme. Note that in the context of secure multiparty computation with up to t corrupted parties, the threshold of $t+1$ ensures that the corrupted parties (even when combining all t of their shares) can learn nothing.

A secret-sharing scheme consist of two algorithm: The first algorithm, called the sharing algorithm, takes as input the secret s and the parameters $t+1$ and n, and outputs n shares. The second algorithm, called the reconstruction algorithm, takes as input $t+1$ or more shares and outputs a value s. It is required that the reconstruction of shares generated from a value s yields the same value s.

Informally, Shamir's secret-sharing scheme works as follows. Let $\mathbb{F}$ be a finite field of size greater than n and let $s \in \mathbb{F}$. The sharing algorithm defines a polynomial $q(x)$ of degree t in $\mathbb{F}[x]$, such that its constant term is the secret s and all the other coefficients are selected uniformly and independently at random in $\mathbb{F}$.[2] Finally, the shares are defined to be $q(\alpha_i)$ for every $i \in \{1, \ldots, n\}$, where $\alpha_1, \ldots, \alpha_n$ are any n distinct nonzero predetermined values in $\mathbb{F}$. The reconstruction algorithm of this scheme is based on the fact that any $t+1$ points define exactly one polynomial of degree t. Therefore, using interpolation it is possible to efficiently reconstruct the polynomial $q(x)$ given any subset of $t+1$ points $(\alpha_i, q(\alpha_i))$ output by the sharing algorithm. Finally, given $q(x)$ it is possible to simply compute $s = q(0)$. We will actually refer to reconstruction using all n points, even though $t+1$ suffice, since this is the way that we use reconstruction throughout the paper.

In order to see that any subset of t or less shares reveals nothing about s, observe that for every set of t points $(\alpha_i, q(\alpha_i))$ and every possible secret $s' \in \mathbb{F}$, there exists a unique polynomial $q'(x)$ such that $q'(0) = s'$ and $q'(\alpha_i) = q(\alpha_i)$. Since the polynomial is chosen randomly by the sharing algorithm, there is the same likelihood that the underlying polynomial is $q(x)$ (and so the secret is s) and that the polynomial is $q'(x)$ (and so the secret is s'). We now formally describe the scheme.

Shamir's $(t+1)$-out-of-n Secret-Sharing Scheme Let $\mathbb{F}$ be a finite field of order greater than n, let $\alpha_1, \ldots, \alpha_n$ be any *distinct nonzero* elements of $\mathbb{F}$, and denote $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$. For a polynomial q Let $\mathsf{eval}_{\vec{\alpha}}(q(x)) = (q(\alpha_1), \ldots, q(\alpha_n))$.

- **The sharing algorithm for** $\alpha_1, \ldots, \alpha_n$: Let $\mathsf{share}_{\vec{\alpha}}(s, t+1)$ be the algorithm that receives for input s and $t+1$ where $s \in \mathbb{F}$ and $t < n$. Then, $\mathsf{share}_{\vec{\alpha}}$ chooses t random values $q_1, \ldots q_t \in_R \mathbb{F}$, independently and uniformly distributed in $\mathbb{F}$, and defines the polynomial:

$$q(x) = s + q_1 x + \cdots q_t x^t$$

  where all calculations are in the field $\mathbb{F}$. Finally, $\mathsf{share}_{\vec{\alpha}}$ outputs $\mathsf{eval}_{\vec{\alpha}}(q(x)) = (q(\alpha_1), \ldots, q(\alpha_n))$, where $q(\alpha_i)$ is the share of party P_i.
- **The reconstruction algorithm**: Algorithm $\mathsf{reconstruct}_{\vec{\alpha}}(\beta_1, \ldots, \beta_n)$ finds the unique polynomial $q(x)$ of degree t such that for every $i = 1, \ldots, n$ it holds that $q(\alpha_i) = \beta_i$, when such a polynomial exists (this holds as long as $\beta_1, \ldots, \beta_n$ all lie on a single polynomial). The algorithm then outputs the coefficients of the polynomial $q(x)$ (note that the original secret can be obtained by simply computing $s = q(0)$).

By the above notation, observe that for every polynomial $q(x)$ of degree $t < n$, it holds that

$$\mathsf{reconstruct}_{\vec{\alpha}}(\mathsf{eval}_{\vec{\alpha}}(q(x))) = q(x). \tag{3.1}$$

[2]Throughout, when we refer to a polynomial of degree t, we mean of degree *at most* t.

Notation Let $\mathcal{P}^{s,t}$ be the set of all polynomials with degree less than or equal to t with constant term s. Observe that for every two values $s, s' \in \mathbb{F}$, it holds that $|\mathcal{P}^{s,t}| = |\mathcal{P}^{s',t}| = |\mathbb{F}|^t$.

3.2. *Basic Properties*

In this section, we state some basic properties of Shamir's secret sharing scheme (the proofs of these claims are standard and appear in [1] for completeness).

In the protocol for secure computation, a dealer hides a secret s by choosing a polynomial $f(x)$ at random from $\mathcal{P}^{s,t}$, and each party P_i receives a share, which is a point $f(\alpha_i)$. In this context, the adversary controls a subset of at most t parties, and thus receives at most t shares. We now show that any subset of at most t shares does not reveal *any* information about the secret. In Sect. 3.1, we explained intuitively why the above holds. This is formalized in the following claim that states that for every subset $I \subset [n]$ with $|I| \leq t$ and every two secrets s, s', the distribution over the shares seen by the parties P_i $(i \in I)$ when s is shared is identical to when s' is shared.

Claim 3.1. *For any set of distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, any pair of values $s, s' \in \mathbb{F}$, any subset $I \subset [n]$ where $|I| = \ell \leq t$, and every $\vec{y} \in \mathbb{F}^\ell$ it holds that:*

$$\Pr_{f(x) \in_R \mathcal{P}^{s,t}} \left[\vec{y} = \left(\{f(\alpha_i)\}_{i \in I}\right)\right] = \Pr_{g(x) \in_R \mathcal{P}^{s',t}} \left[\vec{y} = \left(\{g(\alpha_i)\}_{i \in I}\right)\right] = \frac{1}{|\mathbb{F}|^\ell}$$

where $f(x)$ and $g(x)$ are chosen uniformly and independently from $\mathcal{P}^{s,t}$ and $\mathcal{P}^{s',t}$, respectively.

As a corollary, we have that any $\ell \leq t$ points on a random polynomial are uniformly distributed in the field $\mathbb{F}$. This follows immediately from Claim 3.1 because stating that every $\vec{y}$ appears with probability $1/|\mathbb{F}|^\ell$ is equivalent to stating that the shares are uniformly distributed. That is:

Corollary 3.2. *For any secret $s \in \mathbb{F}$, any set of distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, and any subset $I \subset [n]$ where $|I| = \ell \leq t$, it holds that $\{\{f(\alpha_i)\}_{i \in I}\} \equiv \{U_{\mathbb{F}}^{(1)}, \ldots, U_{\mathbb{F}}^{(\ell)}\}$, where $f(x)$ is chosen uniformly at random from $\mathcal{P}^{s,t}$ and $U_{\mathbb{F}}^{(1)}, \ldots, U_{\mathbb{F}}^{(\ell)}$ are ℓ independent random variables that are uniformly distributed over $\mathbb{F}$.*

Multiple Polynomials In the protocol for secure computation, parties hide secrets and distribute them using Shamir's secret-sharing scheme. As a result, the adversary receives $m \cdot |I|$ shares, $\{f_1(\alpha_i), \ldots, f_m(\alpha_i)\}_{i \in I}$, for some value m. The secrets $f_1(0), \ldots, f_m(0)$ may not be independent. We therefore need to show that the shares that the adversary receives for all secrets do not reveal any information about any of the secrets. Intuitively, this follows from the fact that Claim 3.1 is stated for *any* two secrets s, s', and in particular for two secrets that are known and may be related. The following claim can be proven using standard facts from probability:

Claim 3.3. *For any $m \in \mathbb{N}$, any set of nonzero distinct values $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, any two sets of secrets $(a_1, \ldots, a_m) \in \mathbb{F}^m$ and $(b_1, \ldots, b_m) \in \mathbb{F}^m$, and any subset $I \subset [n]$ of size $|I| \leq t$, it holds that:*

$$\left\{ \{(f_1(\alpha_i), \ldots, f_m(\alpha_i))\}_{i \in I} \right\} \equiv \left\{ \{(g_1(\alpha_i), \ldots, g_m(\alpha_i))\}_{i \in I} \right\}$$

where for every j, $f_j(x)$, $g_j(x)$ are chosen uniformly at random from $\mathcal{P}^{a_j,t}$ and $\mathcal{P}^{b_j,t}$, respectively.

Hiding the Leading Coefficient In Shamir's secret-sharing scheme, the dealer creates shares by constructing a polynomial of degree t, where its constant term is fixed and all the other coefficients are chosen uniformly at random. In Claim 3.1 we showed that any t or fewer points on such a polynomial do not reveal any information about the fixed coefficient which is the constant term.

We now consider this claim when we choose the polynomial differently. In particular, we now fix the *leading* coefficient of the polynomial (i.e., the coefficient of the monomial x^t), and choose all the other coefficients uniformly and independently at random, including the constant term. As in the previous section, it holds that any subset of t or fewer points on such a polynomial do not reveal any information about the fixed coefficient, which in this case is the leading coefficient. We will need this claim for proving the security of one of the subprotocols for the malicious case (in Sect. 6.6).

Let $\mathcal{P}^{\mathsf{lead}}_{s,t}$ be the set of all the polynomials of degree t with *leading* coefficient s. Namely, the polynomials have the structure: $f(x) = a_0 + a_1 x + \cdots a_{t-1} x^{t-1} + s x^t$. The following claim is derived similarly to Corollary 3.2.

Claim 3.4. *For any secret $s \in \mathbb{F}$, any set of distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, and any subset $I \subset [n]$ where $|I| = \ell \leq t$, it holds that:*

$$\left\{ \{f(\alpha_i)\}_{i \in I} \right\} \equiv \left\{ U_{\mathbb{F}}^{(1)}, \ldots, U_{\mathbb{F}}^{(\ell)} \right\}$$

where $f(x)$ is chosen uniformly at random from $\mathcal{P}^{\mathsf{lead}}_{s,t}$ and $U_{\mathbb{F}}^{(1)}, \ldots, U_{\mathbb{F}}^{(\ell)}$ are ℓ independent random variables that are uniformly distributed over $\mathbb{F}$.

3.3. *Matrix Representation*

In this section, we present a useful representation for polynomial evaluation. We define the Vandermonde matrix for the values $\alpha_1, \ldots, \alpha_n$. As is well known, the evaluation of a polynomial at $\alpha_1, \ldots, \alpha_n$ can be obtained by multiplying the associated Vandermonde matrix with the vector containing the polynomial coefficients.

Definition 3.5. (*Vandermonde matrix for* $(\alpha_1, \ldots, \alpha_n)$) Let $\alpha_1, \ldots, \alpha_n$ be n distinct nonzero elements in $\mathbb{F}$. The Vandermonde matrix $V_{\vec{\alpha}}$ for $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ is the $n \times n$ matrix over $\mathbb{F}$ defined by $V_{\vec{\alpha}}[i, j] \stackrel{\text{def}}{=} (\alpha_i)^{j-1}$. That is,

$$V_{\vec{\alpha}} \stackrel{\text{def}}{=} \begin{pmatrix} 1 & \alpha_1 & \dots & (\alpha_1)^{n-1} \\ 1 & \alpha_2 & \dots & (\alpha_2)^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \alpha_n & \dots & (\alpha_n)^{n-1} \end{pmatrix} \tag{3.2}$$

The following fact from linear algebra will be of importance to us:

Fact 3.6. *Let $\vec{\alpha} = (\alpha_1, \dots, \alpha_n)$, where all α_i are distinct and nonzero. Then, $V_{\vec{\alpha}}$ is invertible.*

Matrix Representation of Polynomial Evaluations Let $V_{\vec{\alpha}}$ be the Vandermonde matrix for $\vec{\alpha}$ and let $q = q_0 + q_1 x + \cdots + q_t x^t$ be a polynomial where $t < n$. Define the vector $\vec{q}$ of length n as follows: $\vec{q} \stackrel{\text{def}}{=} (q_0, \dots q_t, 0, \dots, 0)$. Then, it holds that:

$$V_{\vec{\alpha}} \cdot \vec{q} = \begin{pmatrix} 1 & \alpha_1 & \dots & (\alpha_1)^{n-1} \\ 1 & \alpha_2 & \dots & (\alpha_2)^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & \alpha_n & \dots & (\alpha_n)^{n-1} \end{pmatrix} \cdot \begin{pmatrix} q_0 \\ \vdots \\ q_t \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} q(\alpha_1) \\ \vdots \\ \vdots \\ q(\alpha_n) \end{pmatrix}$$

which is the evaluation of the polynomial $q(x)$ on the points $\alpha_1, \dots, \alpha_n$.

4. The Protocol for Semi-honest Adversaries

4.1. *Overview*

We now provide a high-level overview of the protocol for t-privately computing any deterministic functionality in the presence of a semi-honest adversary who controls up to at most $t < n/2$ parties. Let $\mathbb{F}$ be a finite field of size greater than n and let $f : \mathbb{F}^n \to \mathbb{F}^n$ be the functionality that the parties wish to compute. Note that we assume that each party's input and output is a *single field element*. This is only for the sake of clarity of exposition, and the modifications to the protocol for the general case are straightforward. Let C be an *arithmetic circuit* with fan-in of 2 that computes f. We assume that all arithmetic operations in the circuit are carried out over $\mathbb{F}$. In addition, we assume that the arithmetic circuit C consists of three types of gates: *addition* gates, *multiplication* gates, and *multiplication-by-a-constant* gates. Recall that since a circuit is acyclic, it is possible to sort the wires so that for every gate the input wires come before the output wires.

The protocol works by having the parties jointly propagate values through the circuit from the input wires to the output wires, so that at each stage of the computation the parties obtain Shamir-shares of the value on the wire that is currently being computed. In more detail, the protocol has three phases:

- **The input-sharing stage**: In this stage, each party creates shares of its input using Shamir's secret-sharing scheme using threshold $t+1$ (for a given $t < n/2$), and distributes the shares among the parties.
- **The circuit emulation stage**: In this stage, the parties jointly emulate the computation of the circuit C, gate by gate. In each step, the parties compute shares of the output of a given gate, based on the shares of the inputs to that gate that they already have. The actions of the parties in this stage depends on the type of gate being computed:
  1. *Addition gate* Given shares of the input wires to the gate, the output is computed without any interaction by each party simply adding their local shares together. Let the inputs to the gate be a and b and let the shares of the parties be defined by two degree-t polynomials $f_a(x)$ and $f_b(x)$ (meaning that each party P_i holds $f_a(\alpha_i)$ and $f_b(\alpha_i)$ where $f_a(0) = a$ and $f_b(0) = b$). Then the polynomial $f_{a+b}(x)$ defined by shares $f_{a+b}(\alpha_i) = f_a(\alpha_i) + f_b(\alpha_i)$, for every i, is a degree-t polynomial with constant term $a+b$. Thus, each party simply locally adds its own shares $f_a(\alpha_i)$ and $f_b(\alpha_i)$ together, and the result is that the parties hold legal shares of the sum of the inputs, as required.
  2. *Multiplication-by-a-constant gate* This type of gate can also be computed without any interaction. Let the input to the gate be a and let $f_a(x)$ be the t-degree polynomial defining the shares, as above. The aim of the parties is to obtain shares of the value $c \cdot a$, where c is the constant of the gate. Then, each party P_i holding $f_a(\alpha_i)$ simply defines its output share to be $f_{c\cdot a}(\alpha_i) = c \cdot f_a(\alpha_i)$. It is clear that $f_{c\cdot a}(x)$ is a degree-t polynomial with constant term $c \cdot a$, as required.
  3. *Multiplication gate* As in (1) above, let the inputs be a and b, and let $f_a(x)$ and $f_b(x)$ be the polynomials defining the shares. Here, as in the case of an addition gate, the parties can just multiply their shares together and define $h(\alpha_i) = f_a(\alpha_i) \cdot f_b(\alpha_i)$. The constant term of this polynomial is $a \cdot b$, as required. However, $h(x)$ will be of degree $2t$ instead of t; after repeated multiplications the degree will be n or greater and the parties' n shares will not determine the polynomial or enable reconstruction. In addition, $h(x)$ generated in this way is not a "random polynomial" but has a specific structure. For example, $h(x)$ is typically not irreducible (since it can be expressed as the product of $f_a(x)$ and $f_b(x)$), and this may leak information. Thus, local computation does not suffice for computing a multiplication gate. Instead, the parties compute this gate by running an interactive protocol that t-privately computes the multiplication functionality F_{mult}, defined by

$$F_{mult}((f_a(\alpha_1), f_b(\alpha_1)), \ldots, (f_a(\alpha_n), f_b(\alpha_n))) = (f_{ab}(\alpha_1), \ldots, f_{ab}(\alpha_n)) \tag{4.1}$$

     where $f_{ab}(x) \in_R \mathcal{P}^{a\cdot b,t}$ is a random degree-t polynomial with constant term $a \cdot b$.[3]

[3]This definition of the functionality assumes that all of the inputs lie on the polynomials $f_a(x)$, $f_b(x)$ and ignores the case that this does not hold. However, since we are dealing with the semi-honest case here,

- **The output reconstruction stage**: At the end of the computation stage, the parties hold shares of the output wires. In order to obtain the actual output, the parties send their shares to one another and reconstruct the values of the output wires. Specifically, if a given output wire defines output for party P_i, then all parties send their shares of that wire value to P_i.

Organization of This Section In Sect. 4.2, we fully describe the above protocol and prove its security in the F_{mult}-hybrid model. (Recall that in this model, the parties have access to a trusted party who computes F_{mult} for them, and in addition exchange real protocol messages.) We also derive a corollary for t-privately computing any linear function in the plain model (i.e., without any use of the F_{mult} functionality), that is used later in Sect. 4.3.3. Then, in Sect. 4.3, we show how to t-privately compute the F_{mult} functionality for any $t < n/2$. This involves specifying and implementing two functionalities F_{rand}^{2t} and F_{reduce}^{deg}; see the beginning of Sect. 4.3 for an overview of the protocol for t-privately computing F_{mult} and for the definition of these functionalities.

4.2. *Private Computation in the F_{mult}-Hybrid Model*

In this section we present a formal description and proof of the protocol for t-privately computing any deterministic functionality f in the F_{mult}-hybrid model. As we have mentioned, it is assumed that each party has a single input in a known field $\mathbb{F}$ of size greater than n, and that the arithmetic circuit C is over $\mathbb{F}$. See Protocol 4.1 for the description.

We now prove the security of Protocol 4.1. We remark that in the F_{mult}-hybrid model, the protocol is actually t-private for *any* $t < n$. However, as we will see, in order to t-privately compute the F_{mult} functionality, we will need to set $t < n/2$.

Theorem 4.2. *Let $\mathbb{F}$ be a finite field, let $f\colon \mathbb{F}^n \to \mathbb{F}^n$ be an n-ary functionality, and let $t < n$. Then, Protocol 4.1 is t-private for f in the F_{mult}-hybrid model, in the presence of a static semi-honest adversary.*

Proof. Intuitively, the protocol is t-private because the only values that the parties see until the output stage are random shares. Since the threshold of the secret-sharing scheme used is $t + 1$, it holds that no adversary controlling t parties can learn anything. The fact that the view of the adversary can be simulated is due to the fact that t shares of any two possible secrets are identically distributed; see Claim 3.1. This implies that the simulator can generate the shares based on any arbitrary value, and the resulting view is identical to that of a real execution. Observe that this is true until the output stage where the simulator must make the random shares that were used match the actual output of the corrupted parties. This is not a problem because, by interpolation, any set of t shares can be used to define a t-degree polynomial with its constant term being the actual output.

Footnote 3 continued

the inputs are always guaranteed to be correct. This can be formalized using the notion of a partial functionality [20, Sec. 7.2].

Since C computes the functionality f, it is immediate that $\text{OUTPUT}^{\pi}(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$, where π denotes Protocol 4.1. We now proceed to show the existence of a simulator $\mathcal{S}$ as required by Definition 2.1. Before describing the simulator, we present some necessary notation. Our proof works by inductively showing that the partial view of the adversary at every stage is identical in the simulated and real executions. Recall that the view of party P_i is the vector $(x_i, r_i; m_i^1, \ldots, m_i^\ell)$, where x_i is the party's input, r_i its random tape, m_i^k is the kth message that it receives in the execution, and ℓ is the overall number of messages received (in our context here, we let m_i^k equal the series of messages that P_i receives when the parties compute gate G_k). For the sake of clarity, we add to the view of each party the values $\sigma_i^1, \ldots, \sigma_i^\ell$, where σ_i^k equals the shares on the wires that Party P_i holds *after* the parties emulate the computation of gate G_k. That is, we denote

$$\text{VIEW}_i^{\pi}(\vec{x}) = \left(x_i, r_i; m_i^1, \sigma_i^1, \ldots, m_i^\ell, \sigma_i^\ell\right).$$

We stress that since the σ_i^k values can be efficiently computed from the party's input, random tape and incoming messages, the view including the σ_i^k values is equivalent to the view without them, and this is only a matter of notation.

PROTOCOL 4.1 (**t-Private Computation in the F_{mult}-Hybrid Model**).

- **Inputs**: Each party P_i has an input $x_i \in \mathbb{F}$.
- **Auxiliary input**: Each party P_i has an arithmetic circuit C over the field $\mathbb{F}$, such that for every $\vec{x} \in \mathbb{F}^n$ it holds that $C(\vec{x}) = f(\vec{x})$, where $f: \mathbb{F}^n \to \mathbb{F}^n$. The parties also have a description of $\mathbb{F}$ and distinct nonzero values $\alpha_1, \ldots, \alpha_n$ in $\mathbb{F}$.
- **The protocol**:
  1. **The input-sharing stage**: Each party P_i chooses a polynomial $q_i(x)$ uniformly from the set $\mathcal{P}^{x_i,t}$ of all polynomials of degree t with constant term x_i. For every $j \in \{1, \ldots, n\}$, P_i sends party P_j the value $q_i(\alpha_j)$. Each party P_i records the values $q_1(\alpha_i), \ldots, q_n(\alpha_i)$ that it received.
  2. **The circuit emulation stage**: Let $G_1, \ldots, G_\ell$ be a predetermined topological ordering of the gates of the circuit. For $k = 1, \ldots, \ell$ the parties work as follows:

     – *Case 1—G_k is an addition gate*: Let β_i^k and γ_i^k be the shares of input wires held by party P_i. Then, P_i defines its share of the output wire to be $\delta_i^k = \beta_i^k + \gamma_i^k$.

     – *Case 2—G_k is a multiplication-by-a-constant gate with constant c*: Let β_i^k be the share of the input wire held by party P_i. Then, P_i defines its share of the output wire to be $\delta_i^k = c \cdot \beta_i^k$.

     – *Case 3—G_k is a multiplication gate*: Let β_i^k and γ_i^k be the shares of input wires held by party P_i. Then, P_i sends (β_i^k, γ_i^k) to the ideal functionality F_{mult} of Eq. (4.1) and receives back a value δ_i^k. Party P_i defines its share of the output wire to be δ_i^k.
  3. **The output reconstruction stage**: Let $o_1, \ldots, o_n$ be the output wires, where party P_i's output is the value on wire o_i. For every $k = 1, \ldots, n$, denote by $\beta_1^k, \ldots, \beta_n^k$ the shares that the parties hold for wire o_k. Then, each P_i sends P_k the share β_i^k. Upon receiving all shares, P_k computes $\mathsf{reconstruct}_{\vec{\alpha}}(\beta_1^k, \ldots, \beta_n^k)$ and obtains a polynomial $g_k(x)$ (note that $t+1$ of the n shares suffice). P_k then defines its output to be $g_k(0)$.

We are now ready to describe the simulator $\mathcal{S}$. Loosely speaking, $\mathcal{S}$ works by simply sending random shares of arbitrary values until the output stage. Then, in the final output stage $\mathcal{S}$ sends values so that the reconstruction of the shares on the output wires yield the actual output.

The Simulator $\mathcal{S}$

- **Input**: *The simulator receives the inputs and outputs, $\{x_i\}_{i\in I}$ and $\{y_i\}_{i\in I}$ respectively, of all corrupted parties.*
- **Simulation**:

1. Simulating the input-sharing stage:
   (a) *For every $i \in I$, the simulator $\mathcal{S}$ chooses a uniformly distributed random tape for P_i; this random tape and the input x_i fully determines the degree-t polynomial $q'_i(x) \in \mathcal{P}^{x_i,t}$ chosen by P_i in the protocol.*
   (b) *For every $j \notin I$, the simulator $\mathcal{S}$ chooses a random degree-t polynomial $q'_k(x) \in_R \mathcal{P}^{0,t}$ with constant term 0.*
   (c) *The view of the corrupted party P_i in this stage is then constructed by $\mathcal{S}$ to be the set of values $\{q_j(\alpha_i)\}_{j\notin I}$ (i.e., the share sent by each honest P_j to P_i). The view of the adversary $\mathcal{A}$ consists of the view of P_i for every $i \in I$.*
2. Simulating the circuit emulation stage: *For every $G_k \in \{G_1, \ldots, G_\ell\}$*:
   (a) *G_k is an addition gate: Let $\{f_a(\alpha_i)\}_{i\in I}$ and $\{f_b(\alpha_i)\}_{i\in I}$ be the shares of the input wires of the corrupted parties that were generated by $\mathcal{S}$ (initially these are input wires and so the shares are defined by $q'_k(x)$ above). For every $i \in I$, the simulator $\mathcal{S}$ computes $f_a(\alpha_i) + f_b(\alpha_i) = (f_a + f_b)(\alpha_i)$ which defines the shares of the output wire of G_k.*
   (b) *G_k is a multiplication-with-constant gate: Let $\{f_a(\alpha_i)\}_{i\in I}$ be the shares of the input wire and let $c \in \mathbb{F}$ be the constant of the gate. $\mathcal{S}$ computes $c \cdot f_a(\alpha_i) = (c \cdot f_a)(\alpha_i)$ for every $i \in I$ which defines the shares of the output wire of G_k.*
   (c) *G_k is a multiplication gate: $\mathcal{S}$ chooses a degree-t polynomial $f_{ab}(x)$ uniformly at random from $\mathcal{P}^{0,t}$ (irrespective of the shares of the input wires), and defines the shares of the corrupted parties of the output wire of G_k to be $\{f_{ab}(\alpha_i)\}_{i\in I}$. $\mathcal{S}$ adds the shares to the corrupted parties' views.*
3. Simulating the output reconstruction stage: *Let $o_1, \ldots, o_n$ be the output wires. We now focus on the output wires of the corrupted parties. For every $k \in I$, the simulator $\mathcal{S}$ has already defined $|I|$ shares $\{\beta_k^i\}_{i\in I}$ for the output wire o_k. $\mathcal{S}$ thus chooses a random polynomial $g'_k(x)$ of degree t under the following constraints*:
   (a) *$g'_k(0) = y_k$, where y_k is the corrupted P_k's output (the polynomial's constant term is the correct output).*
   (b) *For every $i \in I$, $g'_k(\alpha_i) = \beta_k^i$ (i.e., the polynomial is consistent with the shares that have already been defined).*
   (Note that if $|I| = t$, then the above constraints yield $t + 1$ equations, which in turn fully determine the polynomial $g'_k(x)$. However, if $|I| < t$, then $\mathcal{S}$ can carry out the above by choosing $t - |I|$ additional random points and interpolating.)

Finally, $\mathcal{S}$ adds the shares $\{g'_k(\alpha_1), \ldots, g'_k(\alpha_n)\}$ to the view of the corrupted party P_k.

4. *$\mathcal{S}$ outputs the views of the corrupted parties and halts.*

Denote by $\widetilde{\text{VIEW}}_I^{\pi}(\vec{x})$ the VIEW of the corrupted parties up to the output reconstruction stage (and not including that stage). Likewise, we denote by $\tilde{\mathcal{S}}(I, \vec{x}_I, f_I(\vec{x}))$ the view generated by the simulator up to but not including the output reconstruction stage.

We begin by showing that the partial views of the corrupted parties up to the output reconstruction stage in the real execution and simulation are identically distributed. □

Claim 4.3. *For every $\vec{x} \in \mathbb{F}^n$ and every $I \subset [n]$ with $|I| \leq t$,*

$$\left\{\widetilde{\text{VIEW}}_I^{\pi}(\vec{x})\right\} \equiv \left\{\tilde{\mathcal{S}}(I, \vec{x}_I, f_I(\vec{x}))\right\}$$

Proof. The only difference between the partial views of the corrupted parties in a real and simulated execution is that the simulator generates the shares in the input-sharing stage and in multiplication gates from random polynomials with constant term 0, instead of with the correct value defined by the actual inputs and circuit. Intuitively, the distributions generated are the same since the shares are distributed identically, for every possible secret.

Formally, we construct an algorithm H that receives as input $n - |I| + \ell$ sets of shares: $n - |I|$ sets of shares $\{(i, \beta_i^1)\}_{i \in I}, \ldots, \{(i, \beta_i^{n-|I|})\}_{i \in I}$ and ℓ sets of shares $\{(i, \gamma_i^1)\}_{i \in I}, \ldots, \{(i, \gamma_i^{\ell})\}_{i \in I}$. Algorithm H generates the partial view of the corrupted parties (up until but not including the output reconstruction stage) as follows:

- H uses the jth set of shares $\{\beta_i^j\}_{i \in I}$ as the shares sent by the jth honest party to the corrupted parties in the input-sharing stage (here $j = 1, \ldots, n - |I|$),
- H uses the kth set of shares $\{\gamma_i^k\}_{i \in I}$ are viewed as the shares received by the corrupted parties from F_{mult} in the computation of the k gate G_k, if it is a multiplication gate (here $k = 1, \ldots, \ell$).

Otherwise, H works exactly as the simulator $\mathcal{S}$.

It is immediate that if H receives shares that are generated from random polynomials that all have constant term 0, then the generated view is *exactly* the same as the partial view generated by $\mathcal{S}$. In contrast, if H receives shares that are generated from random polynomials that have constant terms as determined by the inputs and circuit (i.e., the shares β_i^j are generated using the input of the jth honest party, and the shares γ_i^k are generated using the value on the output wire of G_k which is fully determined by the inputs and circuit), then the generated view is *exactly* the same as the partial view in a real execution. This is due to the fact that all shares are generated using the correct values, like in a real execution. By Claim 3.1, these two sets of shares are identically distributed and so the two types of views generated by H are identically distribution; that is, the partial views from the simulated and real executions are identically distributed. □

It remains to show that the output of the simulation after the output reconstruction stage is identical to the view of the corrupted parties in a real execution. For simplicity, we assume that the output wires appear immediately after multiplication gates (otherwise, they are fixed functions of these values).

Before proving this, we prove a claim that describes the processes of the real execution and simulation in a more abstract way. The aim of the claim is to prove that the process carried out by the simulator in the output reconstruction stage yields the same distribution as in a protocol execution. We first describe two processes and prove that they yield the same distribution, and later show how these are related to the real and simulation processes.

Random variable $X(s)$	Random variable $Y(s)$
(1) Choose $q(x) \in_R \mathcal{P}^{s,t}$	(1) Choose $q'(x) \in_R \mathcal{P}^{0,t}$
(2) $\forall i \in I$, set $\beta_i = q(\alpha_i)$	(2) $\forall i \in I$, set $\beta'_i = q'(\alpha_i)$
(3) –	(3) Choose $r(x) \in_R \mathcal{P}^{s,t}$ s.t. $\forall i \in I\ r(\alpha_i) = \beta'_i$
(4) Output $q(x)$	(4) Output $r(x)$

Observe that in $Y(s)$, first the polynomial $q'(x)$ is chosen with constant term 0, and then $r(x)$ is chosen with constant term s, subject to it agreeing with q' on $\{\alpha_i\}_{i \in I}$.

Claim 4.4. *For every $s \in \mathbb{F}$, it holds that $\{X(s)\} \equiv \{Y(s)\}$.*

Intuitively, this follows from the fact that the points $\{q(\alpha_i)\}_{i \in i}$ are distributed identically to $\{q'(\alpha_i)\}_{i \in I}$. The formally proof of the claim follows from a standard probabilistic argument, and appears in [1].

The random variables $X(s)$ and $Y(s)$ can be extended to $X(\vec{s})$ and $Y(\vec{s})$ for any $\vec{s} \in \mathbb{F}^m$ (for some $m \in \mathbb{N}$); the proof of the analogous claim then follows. From this claim, we get:

Claim 4.5. *If $\left\{\widetilde{\text{VIEW}}_I^{\pi}(\vec{x})\right\} \equiv \left\{\tilde{\mathcal{S}}(I, \vec{x}_I, f_I(\vec{x}))\right\}$, then $\left\{\text{VIEW}_I^{\pi}(\vec{x})\right\} \equiv \{\mathcal{S}(I, \vec{x}_I, f_I(\vec{x}))\}$.*

Proof. In the output reconstruction stage, for every $k \in I$, the corrupted parties receive the points $g_k(\alpha_1), \ldots, g_k(\alpha_n)$ in the real execution, and the points $g'_k(\alpha_1), \ldots, g'_k(\alpha_n)$ in the simulation. Equivalently, we can say that the corrupted parties receive the polynomials $\{g_k(x)\}_{k \in I}$ in a real execution, and the polynomials $\{g'_k(x)\}_{k \in I}$ in the simulation.

In the protocol execution, functionality F_{mult} chooses the polynomial $f_{ab}^{(k)}(x)$ for the output wire of P_k uniformly at random in $\mathcal{P}^{y_k,t}$, and the corrupted parties receive values $\beta_i = f_{ab}^{(k)}(\alpha_i)$ (for every $i \in I$). Finally, as we have just described, in the output stage, the corrupted parties receive the polynomials $f_{ab}^{(k)}(x)$ themselves. Thus, this is the process $X(y_k)$. Extending to all $k \in I$, we have that this is the extended process $X(\vec{s})$ with $\vec{s}$ being the vector containing the corrupted parties' output values $\{y_k\}_{k \in I}$.

In contrast, in the simulation of the multiplication gate leading to the output wire for party P_k, the simulator $\mathcal{S}$ chooses the polynomial $f_{ab}^{(k)}(x)$ uniformly at random in $\mathcal{P}^{0,t}$

(see Step 2c in the specification of $\mathcal{S}$ above), and the corrupted parties receive values $\beta_i = f_{ab}^{(k)}(\alpha_i)$ (for every $i \in I$). Then, in the output stage, $\mathcal{S}$ chose $g'_k(x)$ at random from $\mathcal{P}^{y_k,t}$ under the constraint that $g'_k(\alpha_i) = \beta_i$ for every $i \in I$. Thus, this is the process $Y(y_k)$. Extending to all $k \in I$, we have that this is the extended process $Y(\vec{s})$ with $\vec{s}$ being the vector containing the corrupted parties' output values $\{y_k\}_{k \in I}$. The claim thus follows from Claim 4.4. □

Combining Claims 4.3 and 4.5 we have that $\{\mathcal{S}(I, \vec{x}_I, f_I(\vec{x}))\} \equiv \{\text{VIEW}_I^{\pi}(\vec{x})\}$, as required. □

Privately Computing Linear Functionalities in the Real Model Theorem 4.2 states that every function can be t-privately computed in the F_{mult}-hybrid model, for *any* $t < n$. However, a look at Protocol 4.1 and its proof of security show that F_{mult} is only used for computing multiplication gates in the circuit. Thus, Protocol 4.1 can actually be directly used for privately computing any linear functionality f, since such functionalities can be computed by circuits containing only addition and multiplication-by-constant gates. Furthermore, the protocol is secure for any $t < n$; in particular, no honest majority is needed. This yields the following corollary.

Corollary 4.6. *Let $t < n$. Then, any linear functionality f can be t-privately computed in the presence of a static semi-honest adversary. In particular, the matrix multiplication functionality $F_{mat}^A(\vec{x}) = A \cdot \vec{x}$ for matrix $A \in \mathbb{F}^{n \times n}$ can be t-privately computed in the presence of a static semi-honest adversary.*

Corollary 4.6 is used below in order to compute the degree-reduction functionality, which is used in order to privately compute F_{mult}.

4.3. *Privately Computing the F_{mult} Functionality*

We have shown how to t-privately compute any functionality in the F_{mult}-hybrid model. In order to achieve private computation in the plain model, it remains to show how to privately compute the F_{mult} functionality. We remark that the threshold needed to privately compute F_{mult} is $t < n/2$, and thus the overall threshold for the generic BGW protocol is $t < n/2$. Recall that the F_{mult} functionality is defined as follows:

$$F_{mult}((f_a(\alpha_1), f_b(\alpha_1)), \dots, (f_a(\alpha_n), f_b(\alpha_n))) = \Big(f_{ab}(\alpha_1), \dots, f_{ab}(\alpha_n)\Big)$$

where $f_a(x) \in \mathcal{P}^{a,t}$, $f_b(x) \in \mathcal{P}^{b,t}$, and $f_{ab}(x)$ is a *random* polynomial in $\mathcal{P}^{a \cdot b,t}$.

As we have discussed previously, the simple solution where each party locally multiplies its two shares does not work here, for two reasons. First, the resulting polynomial is of degree $2t$ and not t as required. Second, the resulting polynomial of degree $2t$ is not uniformly distributed amongst all polynomials with the required constant term. Therefore, in order to privately compute the F_{mult} functionality, we first *randomize* the degree-$2t$ polynomial so that it is uniformly distributed, and then reduce its degree to t. That is, F_{mult} is computed according to the following steps:

1. Each party locally multiplies its input shares.

2. The parties run a protocol to generate a random polynomial in $\mathcal{P}^{0,2t}$, and each party receives a share based on this polynomial. Then, each party adds its share of the product (from the previous step) with its share of this polynomial. The resulting shares thus define a polynomial which is uniformly distributed in $\mathcal{P}^{a\cdot b,2t}$.
3. The parties run a protocol to reduce the degree of the polynomial to t, with the result being a polynomial that is uniformly distributed in $\mathcal{P}^{a\cdot b,t}$, as required. This computation uses a t-private protocol for computing *matrix multiplication*. We have already shown how to achieve this in Corollary 4.6.

The randomizing (i.e., selecting a random polynomial in $\mathcal{P}^{0,2t}$) and degree-reduction functionalities for carrying out the foregoing steps are formally defined as follows:

- *The randomization functionality*: The randomization functionality is defined as follows:

$$F_{rand}^{2t}(\lambda, \ldots, \lambda) = (r(\alpha_1), \ldots, r(\alpha_n)),$$

  where $r(x) \in_R \mathcal{P}^{0,2t}$ is random, and λ denotes the empty string. We will show how to t-privately compute this functionality in Sect. 4.3.2.
- *The degree-reduction functionality*: Let $h(x) = h_0 + \cdots + h_{2t}x^{2t}$ be a polynomial, and denote by $\mathsf{trunc}_t(h(x))$ the polynomial of degree t with coefficients $h_0, \ldots, h_t$. That is, $\mathsf{trunc}_t(h(x)) = h_0 + h_1x + \cdots + h_tx^t$ (observe that this is a deterministic functionality). Formally, we define

$$F_{reduce}^{deg}(h(\alpha_1), \ldots, h(\alpha_n)) = (\hat{h}(\alpha_1), \ldots, \hat{h}(\alpha_n))$$

  where $\hat{h}(x) = \mathsf{trunc}_t(h(x))$.
  We will show how to t-privately compute this functionality in Sect. 4.3.3.

4.3.1. *Privately Computing F_{mult} in the $(F_{rand}^{2t}, F_{reduce}^{deg})$-Hybrid Model*

We now prove that F_{mult} is reducible to the functionalities F_{rand}^{2t} and F_{reduce}^{deg}; that is, we construct a protocol that t-privately computes F_{mult} given access to ideal functionalities F_{reduce}^{deg} and F_{rand}^{2t}. The full specification appears in Protocol 4.7.

Intuitively, this protocol is secure since the randomization step ensures that the polynomial defining the output shares is random. In addition, the parties only see shares of the randomized polynomial and its truncation. Since the randomized polynomial is of degree $2t$, seeing $2t$ shares of this polynomial still preserves privacy. Thus, the t shares of the randomized polynomial together with the t shares of the truncated polynomial (which is of degree t), still gives the adversary no information whatsoever about the secret. (This last point is the crux of the proof.)

We therefore have:

Proposition 4.8. *Let $t < n/2$. Then, Protocol 4.7 is t-private for F_{mult} in the $(F_{rand}^{2t}, F_{reduce}^{deg})$-hybrid model, in the presence of a static semi-honest adversary.*

Proof. The parties do not receive messages from other parties in the oracle-aided protocol, whereas they receive messages from the oracles only. Therefore, our simulator only needs to simulate the oracle response messages. Since the F_{mult} functionality is probabilistic, we must prove its security using Definition 2.2.

In the real execution of the protocol, the corrupted parties' inputs are $\{f_a(\alpha_i)\}_{i\in I}$ and $\{f_b(\alpha_i)\}_{i\in I}$. Then, in the randomize step of the protocol they receive shares σ_i of a random polynomial of degree $2t$ with constant term 0. Denoting this polynomial by $r(x)$, we have that the corrupted parties receive the values $\{r(\alpha_i)\}_{i\in I}$. Next, the parties invoke the functionality F_{reduce}^{deg} and receive back the values δ_i (these are points of the polynomial $\mathsf{trunc}_t(f_a(x)\cdot f_b(x)+r(x))$). These values are actually the parties' outputs, and thus the simulator must make the output of the call to F_{reduce}^{deg} be the shares $\{\delta_i\}_{i\in I}$ of the corrupted parties outputs.

The Simulator $\mathcal{S}$

- **Input**: *The simulator receives as input I, the inputs of the corrupted parties $\{(\beta_i,\gamma_i)\}_{i\in I}$, and their outputs $\{\delta_i\}_{i\in I}$.*
- **Simulation**:
  - *$\mathcal{S}$ chooses $|I|$ values uniformly and independently at random, $\{v_i\}_{i\in I}$.*
  - *For every $i \in I$, the simulator defines the view of the party P_i to be: $(\beta_i,\gamma_i,v_i,\delta_i)$, where (β_i,γ_i) represents P_i's input, v_i represents P_i's oracle response from F_{rand}^{2t}, and δ_i represents P_i's oracle response from F_{reduce}^{deg}.*

We now proceed to prove that the joint distribution of the output of all the parties, together with the view of the corrupted parties is distributed identically to the output of all parties as computed from the functionality F_{mult} and the output of the simulator. We first show that the outputs of all parties are distributed identically in both cases. Then, we show that the view of the corrupted parties is distributed identically, conditioned on the values of the outputs (and inputs) of all parties.

The Outputs Since the inputs and outputs of all the parties lie on the same polynomials, it is enough to show that the polynomials are distributed identically. Let $f_a(x)$, $f_b(x)$ be the input polynomials. Let $r(x)$ be the output of the F_{rand}^{2t} functionality. Finally, denote the truncated result by $\hat{h}(x) \stackrel{\text{def}}{=} \mathsf{trunc}(f_a(x)\cdot f_b(x)+r(x))$.

In the real execution of the protocol, the parties output shares of the polynomial $\hat{h}(x)$. From the way $\hat{h}(x)$ is defined, it is immediate that $\hat{h}(x)$ is a degree-t polynomial that is uniformly distributed in $\mathcal{P}^{a\cdot b,t}$. (In order to see that it is uniformly distributed,

---

PROTOCOL 4.7 (**t-Privately Computing F_{mult}**).

- **Input**: Each party P_i holds values β_i,γ_i, such that $\mathsf{reconstruct}_{\vec{\alpha}}(\beta_1,\ldots,\beta_n)\in\mathcal{P}^{a,t}$ and $\mathsf{reconstruct}_{\vec{\alpha}}(\gamma_1,\ldots,\gamma_n)\in\mathcal{P}^{b,t}$ for some $a,b\in\mathbb{F}$.
- **The protocol**:
  1. Each party locally computes $s_i=\beta_i\cdot\gamma_i$.
  2. **Randomize**: Each party P_i sends λ to F_{rand}^{2t} (formally, it writes λ on its oracle tape for F_{rand}^{2t}). Let σ_i be the oracle response for party P_i.
  3. **Reduce the degree**: Each party P_i sends $(s_i+\sigma_i)$ to F_{reduce}^{deg}. Let δ_i be the oracle response for P_i.
- **Output**: Each party P_i outputs δ_i.

---

observe that with the exception of the constant term, all the coefficients of the degree-$2t$ polynomial $f_a(x) \cdot f_b(x) + r(x)$ are random. Thus the coefficients of $x, \ldots, x^t$ in $\hat{h}(x)$ are random, as required.)

Furthermore, the functionality F_{mult} return shares for a random polynomial of degree t with constant term $f_a(0) \cdot f_b(0) = a \cdot b$. Thus, the outputs of the parties from a real execution and from the functionality are distributed identically.

The View of the Corrupted Parties We show that the view of the corrupted parties in the real execution and the simulation are distributed identically, given the inputs and outputs of all parties. Observe that the inputs and outputs define the polynomials $f_a(x)$, $f_b(x)$ and $f_{ab}(x)$. Now, the view that is output by the simulator is

$$\left\{ \{ f_a(\alpha_i), f_b(\alpha_i), v_i, f_{ab}(\alpha_i) \}_{i \in I} \right\}$$

where all the v_i values are uniformly distributed in $\mathbb{F}$, and independent of $f_a(x)$, $f_b(x)$ and $f_{ab}(x)$. It remains to show that in a protocol execution the analogous values—which are the outputs received by the corrupted parties from F_{rand}^{2t}—are also uniformly distributed and independent of $f_a(x)$, $f_b(x)$ and $\hat{h}(x)$ (where $\hat{h}(x)$ is distributed identically to a random $f_{ab}(x)$, as already shown above).

In order to prove this, it suffices to prove that for every vector $\vec{y} \in \mathbb{F}^{|I|}$,

$$\Pr\left[\vec{r} = \vec{y} \mid f_a(x), f_b(x), \hat{h}(x) \right] = \frac{1}{|\mathbb{F}|^{|I|}} \tag{4.2}$$

where $\vec{r} = (r(\alpha_{i_1}), \ldots, r(\alpha_{i_{|I|}}))$ for $I = \{i_1, \ldots, i_{|I|}\}$; that is, $\vec{r}$ is the vector of outputs from F_{rand}^{2t}, computed from the polynomial $r(x) \in_R \mathcal{P}^{0,2t}$, that are received by the corrupted parties.

We write $r(x) = r_1(x) + x^t \cdot r_2(x)$, where $r_1(x) \in_R \mathcal{P}^{0,t}$ and $r_2(x) \in_R \mathcal{P}^{0,t}$. In addition, we write $f_a(x) \cdot f_b(x) = h_1(x) + x^t \cdot h_2(x)$, where $h_1(x) \in \mathcal{P}^{ab,t}$ and $h_2(x) \in \mathcal{P}^{0,t}$. Observe that:

$$\begin{aligned} \hat{h}(x) &= \mathsf{trunc}\left(f_a(x) \cdot f_b(x) + r(x) \right) \\ &= \mathsf{trunc}\left(h_1(x) + r_1(x) + x^t \cdot (h_2(x) + r_2(x)) \right) = h_1(x) + r_1(x) \end{aligned}$$

where the last equality holds since the constant term of both $h_2(x)$ and $r_2(x)$ is 0. Rewriting Eq. (4.2), we need to prove that for every vector $\vec{y} \in \mathbb{F}^{|I|}$,

$$\Pr\left[\vec{r} = \vec{y} \mid f_a(x), f_b(x), h_1(x) + r_1(x) \right] = \frac{1}{|\mathbb{F}|^{|I|}}$$

where the kth element r_k of $\vec{r}$ is $r_1(\alpha_{i_k}) + (\alpha_{i_k})^t \cdot r_2(\alpha_{i_k})$. The claim follows since $r_2(x)$ is random and independent of $f_a(x)$, $f_b(x)$, $h_1(x)$ and $r_1(x)$. Formally, for any given $y_k \in \mathbb{F}$, the equality $y_k = r_1(\alpha_{i_k}) + (\alpha_{i_k})^t \cdot r_2(\alpha_{i_k})$ holds if and only if $r_2(\alpha_{i_k}) = (\alpha_{i_k})^{-t} \cdot (y_k - r_1(\alpha_{i_k}))$. Since α_{i_k}, y_k and $r_1(\alpha_{i_k})$ are all fixed by the conditioning, the probability follows from Claim 3.1.

We conclude that the view of the corrupted parties is identically distributed to the output of the simulator, when conditioning on the inputs and outputs of all parties. □

4.3.2. *Privately Computing F^{2t}_{rand} in the Plain Model*

Recall that the randomization functionality is defined as follows:

$$F^{2t}_{rand}(\lambda, \dots, \lambda) = (r(\alpha_1), \dots, r(\alpha_n)), \tag{4.3}$$

where $r(x) \in_R \mathcal{P}^{0,2t}$, and λ denotes the empty string. The protocol for implementing the functionality works as follows. Each party P_i chooses a random polynomial $q_i(x) \in_R \mathcal{P}^{0,2t}$ and sends the share $q_i(\alpha_j)$ to every party P_j. Then, each party P_i outputs $\delta_i = \sum_{k=1}^{n} q_k(\alpha_i)$. Clearly, the shares $\delta_1, \dots, \delta_n$ define a polynomial with constant term 0, because all the polynomials in the sum have a zero constant term. Furthermore, the sum of these random $2t$-degree polynomials is a random polynomial in $\mathcal{P}^{0,2t}$, as required. See Protocol 4.9 for a formal description.

PROTOCOL 4.9 (Privately Computing F^{2t}_{rand}).

- **Input**: The parties do not have inputs for this protocol.
- **The protocol**:
  - Each party P_i chooses a random polynomial $q_i(x) \in_R \mathcal{P}^{0,2t}$. Then, for every $j \in \{1, \dots, n\}$ it sends $s_{i,j} = q_i(\alpha_j)$ to party P_j.
  - Each party P_i receives $s_{1,i}, \dots, s_{n,i}$ and computes $\delta_i = \sum_{j=1}^{n} s_{j,i}$.
- **Output**: Each party P_i outputs δ_i.

We now prove that Protocol 4.9 is t-private for F^{2t}_{rand}.

Claim 4.10. *Let $t < n/2$. Then, Protocol 4.9 is t-private for the F^{2t}_{rand} functionality, in the presence of a static semi-honest adversary.*

Proof. Intuitively, the protocol is secure because the only messages that the parties receive are random shares of polynomials in $\mathcal{P}^{0,2t}$. The simulator can easily simulate these messages by generating the shares itself. However, in order to make sure that the view of the corrupted parties is consistent with the actual output provided by the functionality, the simulator chooses the shares so that their sum equals δ_i, the output provided by the functionality to each P_i.

The Simulator $\mathcal{S}$

- **Input**: *The simulator receives as input I and the outputs of the corrupted parties $\{\delta_i\}_{i \in I}$.*
- **Simulation**:
  1. *Fix $\ell \notin I$*
  2. *$\mathcal{S}$ chooses $n-1$ random polynomials $q'_j(x) \in \mathcal{P}^{0,2t}$ for every $j \in [n] \setminus \{\ell\}$. Note that for $i \in I$, this involves setting the random tape of P_i so that it results in it choosing $q'_i(x)$.*

3. *$\mathcal{S}$ sets the values of the remaining polynomial $q'_\ell(x)$ on the points $\{\alpha_i\}_{i\in I}$ by computing $q'_\ell(\alpha_i) = \delta_i - \sum_{j\neq\ell} q'_j(\alpha_i)$ for every $i \in I$.*
4. *$\mathcal{S}$ sets the incoming messages of corrupted party P_i in the protocol to be $(q'_1(\alpha_i), \ldots, q'_n(\alpha_i))$; observe that all of these points are defined.*

- **Output**: *$\mathcal{S}$ sets the view of each corrupted P_i $(i \in I)$ to be the empty input λ, the random tape determined in Step (2) of the simulation, and the incoming messages determined in Step (4).*

We now show that the view of the adversary (containing the views of all corrupted parties) and the output of all parties in a real execution is distributed identically to the output of the simulator and the output of all parties as received from the functionality in an ideal execution.

In order to do this, consider an fictitious simulator $\mathcal{S}'$ who receives the polynomial $r(x)$ instead of the points $\{\delta_i = r(\alpha_i)\}_{i\in I}$. Simulator $\mathcal{S}'$ works in exactly the same way as $\mathcal{S}$ except that it fully defines the remaining polynomial $q'_\ell(x)$ (and not just its values on the points $\{\alpha_i\}_{i\in I}$) by setting $q'_\ell(x) = r(x) - \sum_{j\neq\ell} q'_j(x)$. Then, $\mathcal{S}'$ computes the values $q'_\ell(\alpha_i)$ for every $i \in I$ from $q'_\ell(x)$. The only difference between the simulator $\mathcal{S}$ and the fictitious simulator $\mathcal{S}'$ is with respect to the value of the polynomial $q'_\ell(x)$ on points outside of $\{\alpha_i\}_{i\in I}$. The crucial point to notice is that $\mathcal{S}$ does *not* define these points differently to $\mathcal{S}'$; rather $\mathcal{S}$ does not define them at all. That is, the simulation does not require $\mathcal{S}$ to determine the value of $q'_\ell(x)$ on points outside of $\{\alpha_i\}_{i\in I}$, and so the distributions are identical.

Finally observe that the output distribution generated by $\mathcal{S}'$ is identical to the output of a real protocol. This holds because in a real protocol execution random polynomials $q_1(x), \ldots, q_n(x)$ are chosen and the output points are derived from $\sum_{j=1}^n q_j(x)$, whereas in the fictitious simulation with $\mathcal{S}'$ the order is just reversed; i.e., first $r(x)$ is chosen at random and then $q'_1(x), \ldots, q'_n(x)$ are chosen at random under the constraint that their sum equals $r(x)$. Note that this uses the fact that $r(x)$ is randomly chosen. □

4.3.3. *Privately Computing F_{reduce}^{deg} in the Plain Model*

Recall that the F_{reduce}^{deg} functionality is defined by

$$F_{reduce}^{deg}(h(\alpha_1), \ldots, h(\alpha_n)) = (\hat{h}(\alpha_1), \ldots, \hat{h}(\alpha_n))$$

where $\hat{h}(x) = \mathsf{trunc}_t(h(x))$ is the polynomial $h(x)$ truncated to degree t (i.e., the polynomial with coefficients $h_0, \ldots, h_t$). We begin by showing that in order to transform a vector of shares of the polynomial $h(x)$ to shares of the polynomial $\mathsf{trunc}_t(h(x))$, it suffices to multiply the input shares by a certain matrix of constants.

Claim 4.11. *Let $t < n/2$. Then, there exists a constant matrix $A \in \mathbb{F}^{n\times n}$ such that for every degree-$2t$ polynomial $h(x) = \sum_{j=0}^{2t} h_j \cdot x^j$ and truncated $\hat{h}(x) = \mathsf{trunc}_t(h(x))$, it holds that:*

$$\left(\hat{h}(\alpha_1), \ldots, \hat{h}(\alpha_n)\right)^T = A \cdot \left(h(\alpha_1), \ldots, h(\alpha_n)\right)^T.$$

Proof. Let $\vec{h} = (h_0, \ldots, h_t, \ldots, h_{2t}, 0, \ldots 0)$ be a vector of length n, and let $V_{\vec{\alpha}}$ be the $n \times n$ Vandermonde matrix for $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$. As we have seen in Sect. 3.3, $V_{\vec{\alpha}} \cdot \vec{h}^T = (h(\alpha_1), \ldots, h(\alpha_n))^T$. Since $V_{\vec{\alpha}}$ is invertible, we have that $\vec{h}^T = V_{\vec{\alpha}}^{-1} \cdot (h(\alpha_1), \ldots, h(\alpha_n))^T$. Similarly, letting $\vec{\hat{h}} = (\hat{h}_0, \ldots, \hat{h}_t, 0, \ldots 0)$ we have that $\left(\hat{h}(\alpha_1), \ldots, \hat{h}(\alpha_n)\right)^T = V_{\vec{\alpha}} \cdot \vec{\hat{h}}^T$.

Now, let $T = \{1, \ldots, t\}$, and let P_T be the linear projection of T; i.e., P_T is an $n \times n$ matrix such that $P_T(i, j) = 1$ for every $i = j \in T$, and $P_T(i, j) = 0$ for all other values. It thus follows that $P_T \cdot \vec{h}^T = \vec{\hat{h}}^T$. Combining all of the above, we have that

$$\left(\hat{h}(\alpha_1), \ldots, \hat{h}(\alpha_n)\right)^T = V_{\vec{\alpha}} \cdot \vec{\hat{h}}^T = V_{\vec{\alpha}} \cdot P_T \cdot \vec{h}^T = V_{\vec{\alpha}} \cdot P_T \cdot V_{\vec{\alpha}}^{-1} \cdot (h(\alpha_1), \ldots, h(\alpha_n))^T .$$

The claim follows by setting $A = V_{\vec{\alpha}} \cdot P_T \cdot V_{\vec{\alpha}}^{-1}$. □

By the above claim it follows that the parties can compute F_{reduce}^{deg} by simply multiplying their shares with the constant matrix A from above. That is, the entire protocol for t-privately computing F_{reduce}^{deg} works by the parties t-privately computing the matrix multiplication functionality $F_{mat}^{A}(\vec{x})$ with the matrix A. By Corollary 4.6 (see the end of Sect. 4.2), $F_{mat}^{A}(\vec{x})$ can be t-privately computed for any $t < n$. Since the entire degree reduction procedure consists of t-privately computing $F_{mat}^{A}(\vec{x})$, we have the following proposition:

Proposition 4.12. *For every $t < n/2$, there exists a protocol that is t-private for F_{reduce}^{deg}, in the presence of a static semi-honest adversary.*

4.4. *Conclusion*

In Sect. 4.3.1 we proved that there exists a t-private protocol for computing the F_{mult} functionality in the $(F_{rand}^{2t}, F_{reduce}^{deg})$-hybrid model, for any $t < n/2$. Then, in Sects. 4.3.2 and 4.3.3 we showed that F_{rand}^{2t} and F_{reduce}^{deg}, respectively, can be t-privately computed (in the plain model) for any $t < n/2$. Finally, in Theorem 4.2 we showed that any n-ary functionality can be privately computed in the F_{mult}-hybrid model, for any $t < n$. Combining the above with the modular sequential composition theorem (described in Sect. 2.3), we conclude that:

Theorem 4.13. *Let $\mathbb{F}$ be a finite field, let $f: \mathbb{F}^n \to \mathbb{F}^n$ be an n-ary functionality, and let $t < n/2$. Then, there exists a protocol that is t-private for f in the presence of a static semi-honest adversary.*

5. Verifiable Secret Sharing (VSS)

5.1. *Background*

Verifiable secret sharing (VSS), defined by Chor et al. [15], is a protocol for sharing a secret in the presence of malicious adversaries. Recall that a secret-sharing scheme (with threshold $t + 1$) is made up of two stages. In the first stage (called *sharing*), the dealer shares a secret so that any $t + 1$ parties can later reconstruct the secret, while any subset of t or fewer parties will learn nothing whatsoever about the secret. In the second stage (called *reconstruction*), a set of $t + 1$ or more parties reconstruct the secret. If we consider Shamir's secret-sharing scheme, much can go wrong if the dealer or some of the parties are malicious (e.g., consider the use of secret sharing in Sect. 4). First, in order to share a secret s, the dealer is supposed to choose a random polynomial $q(\cdot)$ of degree t with $q(0) = s$ and then hand each party P_i its share $q(\alpha_i)$. However, nothing prevents the dealer from choosing a polynomial of higher degree. This is a problem because it means that different subsets of $t + 1$ parties may reconstruct different values. Thus, the shared value is not well defined. Second, in the reconstruction phase each party P_i provides its share $q(\alpha_i)$. However, a corrupted party can provide a different value, thus effectively changing the value of the reconstructed secret, and the other parties have no way of knowing that the provided value is incorrect. Thus, we must use a method that either prevents the corrupted parties from presenting incorrect shares, or ensures that it is possible to reconstruct the correct secret s given $n - t$ correct shares, even if they are mixed together with t incorrect shares (and no one knows which of the shares are correct or incorrect). Note that in the context of multiparty computation, n parties participate in the reconstruction and not just $t + 1$; this is utilized in the following construction.

The BGW protocol for verifiable secret sharing ensures that (for $t < n/3$) the shares received by the honest parties are guaranteed to be $q(\alpha_i)$ for a well-defined degree-t polynomial q, even if the dealer is corrupted. This "secure sharing step" is the challenging part of the protocol. Given such a secure sharing, it is possible to use techniques from the field of error-correcting codes in order to reconstruct q (and thus $q(0) = s$) as long as $n - t$ correct shares are provided and $t < n/3$. This is due to the fact that Shamir's secret-sharing scheme when looked at in this context is exactly a Reed–Solomon code, and Reed–Solomon codes can efficiently correct up to t errors, for $t < n/3$.

5.2. *The Reed–Solomon Code*

We briefly describe the Reed–Solomon code, and its use in our context. First, recall that a linear $[n, k, d]$-code over a field $\mathbb{F}$ of size q is a code of length n (meaning that each codeword is a sequence of n field elements), of dimension k (meaning that there are q^k different codewords), and of distance d (meaning that every two codewords are of Hamming distance at least d from each other).

We are interested in constructing a code of length n, dimension $k = t+1$, and distance $n - t$. The Reed–Solomon code for these parameters is constructed as follows. Let $\mathbb{F}$ be a finite field such that $|\mathbb{F}| > n$, and let $\alpha_1, \ldots, \alpha_n$ be distinct field elements. Let $m = (m_0, \ldots, m_t)$ be a message to be encoded, where each $m_i \in \mathbb{F}$. The encoding of m is as follows:

1. Define a polynomial $p_m(x) = m_0 + m_1 x + \cdots + m_t x^t$ of degree t.
2. Compute the codeword $C(m) = \langle p_m(\alpha_1), \ldots, p_m(\alpha_n)\rangle$.

It is well known that the distance of this code is $n - t$. (In order to see this, recall that for any two different polynomials p_1 and p_2 of degree at most t, there are at most t points α for which $p_1(\alpha) = p_2(\alpha)$. Noting that $m \neq m'$ define different polynomials

$p_m \neq p_{m'}$, we have that $C(m)$ and $C(m')$ agree in at most t places.) Let $d(x, y)$ denote the Hamming distance between words $x, y \in \mathbb{F}^n$. The following is a well-known result from the error-correcting code literature:

Theorem 5.1. *The Reed–Solomon code is a linear* $[n, t+1, n-t]$*-code over* $\mathbb{F}$. *In addition, there exists an efficient decoding algorithm that corrects up to* $\frac{n-t-1}{2}$ *errors. That is, for every* $m \in \mathbb{F}^{t+1}$ *and every* $x \in \mathbb{F}^n$ *such that* $d(x, C(m)) \leq \frac{n-t-1}{2}$, *the decoding algorithm returns* m.

Let $t < n/3$, and so $n \geq 3t+1$. Plugging this into Theorem 5.1, we have that it is possible to efficiently correct up to $\frac{3t+1-t-1}{2} = t$ errors.

Reed–Solomon and Shamir's Secret Sharing Assume that n parties hold shares $\{q(\alpha_i)\}_{i\in[n]}$ of a degree-t polynomial, as in Shamir's secret-sharing scheme. That is, the dealer distributed shares $\{q(\alpha_i)\}_{i\in[n]}$ where $q \in_R \mathcal{P}^{s,t}$ for a secret $s \in \mathbb{F}$. We can view the shares $\langle q(\alpha_1), \ldots, q(\alpha_n)\rangle$ as a Reed–Solomon codeword. Now, in order for the parties to reconstruct the secret from the shares, all parties can just broadcast their shares. Observe that the honest parties provide their correct share $q(\alpha_i)$, whereas the corrupted parties may provide incorrect values. However, since the number of corrupted parties is $t < n/3$, it follows that at most t of the symbols are incorrect. Thus, the Reed–Solomon reconstruction procedure can be run and the honest parties can all obtain the correct polynomial q, and can compute $q(0) = s$.

We conclude that in such a case the corrupted parties cannot effectively cheat in the reconstruction phase. Indeed, even if they provide incorrect values, it is possible for the honest parties to correctly reconstruct the secret (*with probability* 1). Thus, the main challenge in constructing a verifiable secret-sharing protocol is how to force a corrupted dealer to distribute shares that are consistent with some degree-t polynomial.

5.3. *Bivariate Polynomials*

Bivariate polynomials are a central tool used by the BGW verifiable secret-sharing protocol (in the sharing stage). We therefore provide a short background to bivariate polynomials in this section.

A bivariate polynomial of degree t is a polynomial over two variables, *each* of which has degree at most t. Such a polynomial can be written as follows:

$$f(x, y) = \sum_{i=0}^{t} \sum_{j=0}^{t} a_{i,j} \cdot x^i \cdot y^j.$$

We denote by $\mathcal{B}^{s,t}$ the set of all bivariate polynomials of degree t and with constant term s. Note that the number of coefficients of a bivariate polynomial in $\mathcal{B}^{s,t}$ is $(t+1)^2 - 1 = t^2 + 2t$ (there are $(t+1)^2$ coefficients, but the constant term is already fixed to be s).

Recall that when considering *univariate* polynomials, $t+1$ points define a unique polynomial of degree t. In this case, each point is a pair (α_k, β_k) and there exists a unique polynomial f such that $f(\alpha_k) = \beta_k$ for all $t+1$ given points $\{(\alpha_k, \beta_k)\}_{k=1}^{t+1}$. The analogous statement for bivariate polynomials is that $t+1$ univariate polynomials of degree t define a unique bivariate polynomial of degree t; see Claim 5.2 below. For a degree-t bivariate polynomial $S(x, y)$, fixing the y-value to be some α defines a degree-t univariate polynomial $f(x) = S(x, \alpha)$. Likewise, any $t+1$ fixed values $\alpha_1, \ldots, \alpha_{t+1}$ define $t+1$ degree-t univariate polynomials $f_k(x) = S(x, \alpha_k)$. What we show now is that like in the univariate case, this works in the opposite direction as well. Specifically, given $t+1$ values $\alpha_1, \ldots, \alpha_{t+1}$ and $t+1$ degree-t polynomials $f_1(x), \ldots, f_{t+1}(x)$ there exists a unique bivariate polynomial $S(x, y)$ such that $S(x, \alpha_k) = f_k(x)$, for every $k = 1, \ldots, t+1$. This is formalized in the next claim, which is a variant of the classic Lagrange interpolation theorem (a proof can also be found in [1,18]):

Claim 5.2. *Let t be a nonnegative integer, let $\alpha_1, \ldots, \alpha_{t+1}$ be $t+1$ distinct elements in $\mathbb{F}$, and let $f_1(x), \ldots, f_{t+1}(x)$ be $t+1$ polynomials of degree t. Then, there exists a unique bivariate polynomial $S(x, y)$ of degree t such that for every $k = 1, \ldots, t+1$ it holds that $S(x, \alpha_k) = f_k(x)$.*

Verifiable Secret Sharing Using Bivariate Polynomials The verifiable secret-sharing protocol works by embedding a random univariate degree-t polynomial $q(z)$ with $q(0) = s$ into the bivariate polynomial $S(x, y)$. Specifically, $S(x, y)$ is chosen at random under the constraint that $S(0, z) = q(z)$; the values $q(\alpha_1), \ldots, q(\alpha_n)$ are thus the univariate Shamir-shares embedded into $S(x, y)$. Then, the dealer sends each party P_i two univariate polynomials as intermediate shares; these polynomials are $f_i(x) = S(x, \alpha_i)$ and $g_i(y) = S(\alpha_i, y)$. By the definition of these polynomials, it holds that $f_i(\alpha_j) = S(\alpha_j, \alpha_i) = g_j(\alpha_i)$, and $g_i(\alpha_j) = S(\alpha_i, \alpha_j) = f_j(\alpha_i)$. Thus, any two parties P_i and P_j can verify that the univariate polynomials that they received are *pairwise consistent* with each other by checking that $f_i(\alpha_j) = g_j(\alpha_i)$ and $g_i(\alpha_j) = f_j(\alpha_i)$. As we shall see, this prevents the dealer from distributing shares that are not consistent with a single bivariate polynomial. Finally, party P_i defines its output (i.e., "Shamir-share") as $f_i(0) = q(\alpha_i)$, as required.

We begin by proving that pairwise consistency checks as described above suffice for uniquely determining the bivariate polynomial S. Specifically:

Claim 5.3. *Let $K \subseteq [n]$ be a set of indices such that $|K| \geq t+1$, let $\{f_k(x), g_k(y)\}_{k \in K}$ be a set of pairs of degree-t polynomials, and let $\{\alpha_k\}_{k \in K}$ be distinct nonzero elements in $\mathbb{F}$. If for every $i, j \in K$, it holds that $f_i(\alpha_j) = g_j(\alpha_i)$, then there exists a unique bivariate polynomial S of degree-t in both variables such that $f_k(x) = S(x, \alpha_k)$ and $g_k(y) = S(\alpha_k, y)$ for every $k \in K$.*

Proof. Let L be any subset of K of cardinality exactly $t+1$. By Claim 5.2, there exists a *unique* bivariate polynomial $S(x, y)$ of degree-t in both variables, for which $S(x, \alpha_\ell) = f_\ell(x)$ for every $\ell \in L$. We now show if $f_i(\alpha_j) = g_j(\alpha_i)$ for all $i, j \in K$, then for every $k \in K$ it holds that $f_k(x) = S(x, \alpha_k)$ and $g_k(y) = S(\alpha_k, y)$.

By the consistency assumption, for every $k \in K$ and $\ell \in L$ we have that $g_k(\alpha_\ell) = f_\ell(\alpha_k)$. Furthermore, by the definition of S from above we have that $f_\ell(\alpha_k) = S(\alpha_k, \alpha_\ell)$. Thus, for all $k \in K$ and $\ell \in L$ it holds that $g_k(\alpha_\ell) = S(\alpha_k, \alpha_\ell)$. Since both $g_k(y)$ and

$S(\alpha_k, y)$ are degree-t polynomials, and $g_k(\alpha_\ell) = S(\alpha_k, \alpha_\ell)$ for $t+1$ points α_ℓ, it follows that $g_k(y) = S(\alpha_k, y)$ for every $k \in K$.

It remains to show that $f_k(x) = S(x, \alpha_k)$ for all $k \in K$ (this trivially holds for all $k \in L$ by the definition of S from above, but needs to be proven for $k \in K \setminus L$). By consistency, for every $j, k \in K$, we have that $f_k(\alpha_j) = g_j(\alpha_k)$. Furthermore, we have already proven that $g_j(\alpha_k) = S(\alpha_j, \alpha_k)$ for every $j, k \in K$. Therefore, $f_k(\alpha_j) = S(\alpha_j, \alpha_k)$ for every $j, k \in K$, implying that $f_k(x) = S(x, \alpha_k)$ for every $k \in K$ (because they are degree-t polynomials who have the same value on more than t points). This concludes the proof. □

We now proceed to prove a "secrecy lemma" for bivariate polynomial secret sharing. Loosely speaking, we prove that the shares $\{f_i(x), g_i(y)\}_{i \in I}$ (for $|I| \leq t$) that the corrupted parties receive do not reveal any information about the secret s. In fact, we show something much stronger: for every two degree-t polynomials q_1 and q_2 such that $q_1(\alpha_i) = q_2(\alpha_i) = f_i(0)$ for every $i \in I$, the distribution over the shares $\{f_i(x), g_i(y)\}_{i \in I}$ received by the corrupted parties when $S(x, y)$ is chosen based on $q_1(z)$ is identical to the distribution when $S(x, y)$ is chosen based on $q_2(z)$. An immediate corollary of this is that no information is revealed about whether the secret equals $s_1 = q_1(0)$ or $s_2 = q_2(0)$.

Claim 5.4. *Let $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$ be n distinct nonzero values, let $I \subset [n]$ with $|I| \leq t$, and let q_1 and q_2 be two degree-t polynomials over $\mathbb{F}$ such that $q_1(\alpha_i) = q_2(\alpha_i)$ for every $i \in I$. Then,*

$$\left\{ \left\{ (i, S_1(x, \alpha_i), S_1(\alpha_i, y)) \right\}_{i \in I} \right\} \equiv \left\{ \left\{ (i, S_2(x, \alpha_i), S_2(\alpha_i, y)) \right\}_{i \in I} \right\}$$

where $S_1(x, y)$ and $S_2(x, y)$ are degree-t bivariate polynomial chosen at random under the constraints that $S_1(0, z) = q_1(z)$ and $S_2(0, z) = q_2(z)$, respectively.

Proof. We begin by defining probability ensembles $\mathbb{S}_1$ and $\mathbb{S}_2$, as follows:

$$\mathbb{S}_1 = \left\{ \{(i, S_1(x, \alpha_i), S_1(\alpha_i, y))\}_{i \in I} \mid S_1 \in_R \mathcal{B}^{q_1(0),t} \text{ s.t. } S_1(0, z) = q_1(z) \right\}$$
$$\mathbb{S}_2 = \left\{ \{(i, S_2(x, \alpha_i), S_2(\alpha_i, y))\}_{i \in I} \mid S_2 \in_R \mathcal{B}^{q_2(0),t} \text{ s.t. } S_2(0, z) = q_2(z) \right\}$$

Given this notation, an equivalent formulation of the claim is that $\mathbb{S}_1 \equiv \mathbb{S}_2$.

In order to prove that this holds, we first show that for any set of pairs of degree-t polynomials $Z = \{(i, f_i(x), g_i(y))\}_{i \in I}$, the number of bivariate polynomials in the support of $\mathbb{S}_1$ that are consistent with Z equals the number of bivariate polynomials in the support of $\mathbb{S}_2$ that are consistent with Z, where consistency means that $f_i(x) = S(x, \alpha_i)$ and $g_i(y) = S(\alpha_i, y)$.

First note that if there exist $i, j \in I$ such that $f_i(\alpha_j) \neq g_j(\alpha_i)$ then there does not exist any bivariate polynomial in the support of $\mathbb{S}_1$ or $\mathbb{S}_2$ that is consistent with Z. Also, if there exists an $i \in I$ such that $f_i(0) \neq q_1(\alpha_i)$, then once again there is no polynomial from $\mathbb{S}_1$ or $\mathbb{S}_2$ that is consistent (this holds for $\mathbb{S}_1$ since $f_i(0) = S(0, \alpha_i) = q_1(\alpha_i)$ should hold, and it holds similarly for $\mathbb{S}_2$ because $q_1(\alpha_i) = q_2(\alpha_i)$ for all $i \in I$).

Let $Z = \{(i, f_i(x), g_i(y))\}_{i\in I}$ be a set of degree-t polynomials such that for every $i, j \in I$ it holds that $f_i(\alpha_j) = g_j(\alpha_i)$, and in addition for every $i \in I$ it holds that $f_i(0) = q_1(\alpha_i) = q_2(\alpha_i)$. We begin by counting how many such polynomials exist in the support of $\mathbb{S}_1$. We have that Z contains $|I|$ degree-t polynomials $\{f_i(x)\}_{i\in I}$, and recall that $t + 1$ such polynomials $f_i(x)$ fully define a degree-t bivariate polynomial. Thus, we need to choose $t + 1 - |I|$ more polynomials $f_j(x)$ ($j \neq i$) that are consistent with $q_1(z)$ and with $\{g_i(y)\}_{i\in I}$. In order for a polynomial $f_j(x)$ to be consistent in this sense, it must hold that $f_j(\alpha_i) = g_i(\alpha_j)$ for every $i \in I$, and in addition that $f_j(0) = q_1(\alpha_j)$. Thus, for each such $f_j(x)$ that we add, $|I|+1$ values of f_j are already determined. Since the values of f_j at $t + 1$ points determine a degree-t univariate polynomial, it follows that an additional $t - |I|$ points can be chosen in all possible ways and the result will be consistent with Z. We conclude that there exist $\left(|\mathbb{F}|^{t-|I|}\right)^{(t+1-|I|)}$ ways to choose S_1 according to $\mathbb{S}_1$ that will be consistent. (Note that if $|I| = t$ then there is just one way.) The important point here is that the exact same calculation holds for S_2 chosen according to $\mathbb{S}_2$, and thus exactly the same number of polynomials from $\mathbb{S}_1$ are consistent with Z as from $\mathbb{S}_2$.

Now, let $Z = \{(i, f_i(x), g_i(y))\}_{i\in I}$ be a set of $|I|$ pairs of univariate degree-t polynomials. We have already shown that the number of polynomials in the support of $\mathbb{S}_1$ that are consistent with Z equals the number of polynomials in the support of $\mathbb{S}_2$ that are consistent with Z. Since the polynomials S_1 and S_2 (in $\mathbb{S}_1$ and $\mathbb{S}_2$, respectively) are chosen randomly among those consistent with Z, it follows that the probability that Z is obtained is exactly the same in both cases, as required. □

5.4. *The Verifiable Secret-Sharing Protocol*

In the VSS functionality, the dealer inputs a polynomial $q(x)$ of degree t, and each party P_i receives its Shamir-share $q(\alpha_i)$ based on that polynomial.[4] The "verifiable" part is that if q is of degree greater than t, then the parties reject the dealer's shares and output $\perp$. The functionality is formally defined as follows:

FUNCTIONALITY 5.5 (The BGW F_{VSS} functionality).

$$F_{VSS}(q(x), \lambda, \ldots, \lambda) = \begin{cases} (q(\alpha_1), \ldots, q(\alpha_n)) & \text{if } \deg(q) \leq t \\ (\perp, \ldots, \perp) & \text{otherwise} \end{cases}$$

Observe that the secret $s = q(0)$ is only implicitly defined in the functionality; it is however well defined. Thus, in order to share a secret s, the functionality is used by having the dealer first choose a random polynomial $q \in_R \mathcal{P}^{s,t}$ (where $\mathcal{P}^{s,t}$ is the set of all degree-t univariate polynomials with constant term s) and then run F_{VSS} with input $q(x)$.

[4] This is a specific VSS definition that is suited for the BGW protocol. We remark that it is possible to define VSS in a more general and abstract way (like a multiparty "commitment"). However, since we will need to *compute* on the shares $q(\alpha_1), \ldots, q(\alpha_n)$, these values need to be explicitly given in the output.

The Protocol Idea We present the VSS protocol of BGW with the simplification of the complaint phase suggested by Feldman [17]. The protocol uses private point-to-point channels between each pair of parties and an *authenticated* broadcast channel (meaning that the identity of the broadcaster is given). The protocol works by the dealer selecting a random bivariate polynomial $S(x, y)$ of degree t under the constraint that $S(0, z) = q(z)$. The dealer then sends each party P_i two polynomials that are derived from $S(x, y)$: the polynomial $f_i(x) = S(x, \alpha_i)$ and the polynomial $g_i(y) = S(\alpha_i, y)$. As we have shown in Claim 5.4, t pairs of polynomials $f_i(x), g_i(y)$ received by the corrupted parties reveal nothing about the constant term of S (i.e., the secret being shared). In addition, given these polynomials, the parties can verify that they have consistent inputs. Specifically, since $g_i(\alpha_j) = S(\alpha_i, \alpha_j) = f_j(\alpha_i)$, it follows that each pair of parties P_i and P_j can check that their polynomials fulfill $f_i(\alpha_j) = g_j(\alpha_i)$ and $g_i(\alpha_j) = f_j(\alpha_i)$ by sending each other these points. If all of these checks pass, then by Claim 5.3 it follows that all the polynomials are derived from a single bivariate polynomial $S(x, y)$, and thus the sharing is valid and the secret is fully determined.

The problem that arises is what happens if the polynomials are not all consistent; i.e., if P_j receives from P_i values $f_i(\alpha_j), g_i(\alpha_j)$ such that $f_j(\alpha_i) \neq g_i(\alpha_j)$ or $g_j(\alpha_i) \neq f_i(\alpha_j)$. This can happen if the dealer is corrupted, or if P_i is corrupted. In such a case, P_j issues a "complaint" by broadcasting its inconsistent values $(j, i, f_j(\alpha_i), g_j(\alpha_i))$ defined by the shares $f_j(x), g_j(y)$ it received from the dealer. Then, the dealer checks whether these values are correct, and if they are not, then it is required to broadcast the correct polynomials for that complaining party. We stress that in this case the dealer broadcasts the *entire polynomials* $f_j(x)$ and $g_j(y)$ defining P_j's share, and this enables all other parties P_k to verify that these polynomials are consistent with their own shares, thus verifying their validity. Note that if the values broadcast *are* correct (e.g., in the case that the dealer is honest and P_i sent P_j incorrect values), then the dealer does not broadcast P_j's polynomials. This ensures that an honest dealer does not reveal the shares of honest parties.

This strategy is sound since if the dealer is honest, then all honest parties will have consistent values. Thus, the only complaints will be due to corrupted parties complaining falsely (in which case the dealer will broadcast the *corrupted parties* polynomials, which gives them no more information), or due to corrupted parties sending incorrect values to honest parties (in which case the dealer does not broadcast anything, as mentioned). In contrast, if the dealer is not honest, then all honest parties will reject and output $\perp$ unless it resends consistent polynomials to all, thereby guaranteeing that $S(x, y)$ is fully defined again, as required. This complaint resolution must be carried out carefully in order to ensure that security is maintained. We defer more explanation about how this works until after the full specification, given in Protocol 5.6.

The Security of Protocol 5.6 Before we prove that Protocol 5.6 is t-secure for the F_{VSS} functionality, we present an intuitive argument as to why this holds. First, consider the case that the dealer is honest. In this case, all of the polynomials received by the parties are consistent (i.e., for every pair P_i, P_j it holds that $f_i(\alpha_j) = g_j(\alpha_i)$ and $f_j(\alpha_i) = g_i(\alpha_j)$). Thus, an honest party P_j only broadcasts a complaint if a corrupted party sends it incorrect values and the values included in that complaint are known already to the adversary. However, if this occurs then the dealer will *not* send a reveal

PROTOCOL 5.6 (Securely Computing F_{VSS}).

- **Input**: The dealer $D = P_1$ holds a polynomial $q(x)$ of degree at most t (if not, then the honest dealer just aborts at the onset). The other parties $P_2, \ldots, P_n$ have no input.
- **Common input**: The description of a field $\mathbb{F}$ and n nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$.
- **The protocol**:

1. **Round 1 (send shares)—the dealer**:
   (a) The dealer selects a uniformly distributed bivariate polynomial $S(x, y) \in \mathcal{B}^{q(0),t}$, under the constraint that $S(0, z) = q(z)$.
   (b) For every $i \in \{1, \ldots, n\}$, the dealer defines the polynomials $f_i(x) \stackrel{\text{def}}{=} S(x, \alpha_i)$ and $g_i(y) \stackrel{\text{def}}{=} S(\alpha_i, y)$. It then sends to each party P_i the polynomials $f_i(x)$ and $g_i(y)$.
2. **Round 2 (exchange subshares)—each party P_i**:
   (a) Store the polynomials $f_i(x)$ and $g_i(y)$ that were received from the dealer. (If $f_i(x)$ or $g_i(y)$ is of degree greater than t then truncate it to be of degree t.)
   (b) For every $j \in \{1, \ldots, n\}$, send $f_i(\alpha_j)$ and $g_i(\alpha_j)$ to party P_j.
3. **Round 3 (broadcast complaints)—each party P_i**:
   (a) For every $j \in \{1, \ldots, n\}$, let (u_j, v_j) denote the values received from player P_j in Round 2 (these are supposed to be $u_j = f_j(\alpha_i)$ and $v_j = g_j(\alpha_i)$). If $u_j \neq g_i(\alpha_j)$ or $v_j \neq f_i(\alpha_j)$, then broadcast $\mathsf{complaint}(i, j, f_i(\alpha_j), g_i(\alpha_j))$.
   (b) If no parties broadcast a complaint, then every party P_i outputs $f_i(0)$ and halts.
4. **Round 4 (resolve complaints)—the dealer**: For every complaint message received, do the following:
   (a) Upon viewing a message $\mathsf{complaint}(i, j, u, v)$ broadcast by P_i, check that $u = S(\alpha_j, \alpha_i)$ and $v = S(\alpha_i, \alpha_j)$. (Note that if the dealer and P_i are honest, then it holds that $u = f_i(\alpha_j)$ and $v = g_i(\alpha_j)$.) If the above condition holds, then do nothing. Otherwise, broadcast $\mathsf{reveal}(i, f_i(x), g_i(y))$.
5. **Round 5 (evaluate complaint resolutions)—each party P_i**:
   (a) For every $j \neq k$, party P_i marks (j, k) as a joint complaint if it viewed two messages $\mathsf{complaint}(k, j, u_1, v_1)$ and $\mathsf{complaint}(j, k, u_2, v_2)$ broadcast by P_k and P_j, respectively, such that $u_1 \neq v_2$ or $v_1 \neq u_2$. If there exists a joint complaint (j, k) for which the dealer did not broadcast $\mathsf{reveal}(k, f_k(x), g_k(y))$ nor $\mathsf{reveal}(j, f_j(x), g_j(y))$, then go to Step 6 (and do not broadcast consistent). Otherwise, proceed to the next step.
   (b) Consider the set of $\mathsf{reveal}(j, f_j(x), g_j(y))$ messages sent by the dealer (truncating the polynomials to degree t if necessary as in Step 2a):
      i. If there exists a message in the set with $j = i$ then reset the stored polynomials $f_i(x)$ and $g_i(y)$ to the new polynomials that were received, and go to Step 6 (without broadcasting consistent).
      ii. If there exists a message in the set with $j \neq i$ and for which $f_i(\alpha_j) \neq g_j(\alpha_i)$ or $g_i(\alpha_j) \neq f_j(\alpha_i)$, then go to Step 6 (without broadcasting consistent).

   If the set of reveal messages does not contain a message that fulfills either one of the above conditions, then proceed to the next step.
   (c) Broadcast the message consistent.
6. **Output decision (if there were complaints)—each party P_i**: If at least $n - t$ parties broadcast consistent, output $f_i(0)$. Otherwise, output $\perp$.

of the honest party's polynomials (because its values are correct). Furthermore, if any corrupted party P_i broadcasts a complaint with incorrect values (u, v), the dealer can send the correct reveal message (this provides no additional information to the adversary since the reveal message just contains the complainant's shares). In such a case, the check carried out by each honest party P_j in Step 5(b)ii will pass and so every honest party will broadcast consistent. Thus, at least $n - t$ parties broadcast consistent (since there are at least $n - t$ honest parties), and so every honest party P_j outputs $f_j(0) = S(0, \alpha_j) = q(\alpha_j)$, where the last equality is due to the way the dealer chooses $S(x, y)$.

Next, consider the case that the dealer is corrupted. In this case, the honest parties may receive polynomials that are not consistent with each other; that is, honest P_j and P_k may receive polynomials $f_j(x)$, $g_j(y)$ and $f_k(x)$, $g_k(y)$ such that either $f_j(\alpha_k) \neq g_k(\alpha_j)$ or $f_k(\alpha_j) \neq g_j(\alpha_k)$. However, in such a case both honest parties complain, and the dealer must send a valid reveal message (in the sense described below) or no honest party will broadcast consistent. In order for $n - t$ parties to broadcast consistent, there must be at least $(n - t) - t = t + 1$ honest parties that broadcast consistent. This implies that these $t + 1$ or more honest parties all received polynomials $f_j(x)$ and $g_j(y)$ in the first round that are *pairwise consistent* with each other and with all of the "fixed" values in the reveal messages. Thus, by Claim 5.3 the polynomials $f_j(x)$ and $g_j(y)$ of these $t+1$ (or more) parties are all derived from a unique degree-t bivariate polynomial $S(x, y)$, meaning that $f_j(x) = S(x, \alpha_j)$ and $g_j(y) = S(\alpha_j, y)$. (The parties who broadcasted consistent are those that make up the set K in Claim 5.3.)

The above suffices to argue that the polynomials of all the honest parties that broadcast consistent are derived from a unique $S(x, y)$. It remains to show that if at least $t + 1$ honest parties broadcast consistent, then the polynomials of all the other honest parties that do not broadcast consistent are also derived from the same $S(x, y)$. Assume that this is not the case. That is, there exists an honest party P_j such that $f_j(x) \neq S(x, \alpha_j)$ (an analogous argument can be made with respect to $g_j(x)$ and $S(\alpha_j, y)$). Since $f_j(x)$ is of degree-t this implies that $f_j(\alpha_k) = S(\alpha_k, \alpha_j)$ for at most t points α_k. Thus, P_j's points are pairwise consistent with at most t honest parties that broadcast consistent (since for all of these parties $g_k(y) = S(\alpha_k, y)$). This implies that there must have been a joint complaint between P_j and an honest party P_k who broadcasted consistent, and so this complaint must have been resolved by the dealer broadcasting polynomials $f_j(x)$ and $g_j(y)$ such that $f_j(\alpha_k) = g_k(\alpha_j)$ for all P_k who broadcasted consistent (otherwise, they would not have broadcasted consistent). We now proceed to the formal proof.

Theorem 5.7. *Let $t < n/3$. Then, Protocol 5.6 is t-secure for the F_{VSS} functionality in the presence of a static malicious adversary.*

Proof. Let $\mathcal{A}$ be an adversary in the real world. We show the existence of a simulator $\mathcal{SIM}$ such that for any set of corrupted parties I and for all inputs, the output of all parties and the adversary $\mathcal{A}$ in an execution of the real protocol with $\mathcal{A}$ is identical to the outputs in an execution with $\mathcal{SIM}$ in the ideal model. We separately deal with the case that the dealer is honest and the case that the dealer is corrupted. Loosely speaking, when the dealer is honest we show that the honest parties always accept the dealt shares, and in particular that the adversary cannot falsely generate complaints that will interfere with the result. In the case that the dealer is corrupted the proof is more involved and consists of showing that if the dealer resolves complaints so that at least $n - t$ parties broadcast consistent, then this implies that at the end of the protocol all honest parties hold consistent shares, as required. □

Case 1: The Dealer is Honest

In this case in an *ideal execution*, the dealer sends $q(x)$ to the trusted party and each honest party P_j receives $q(\alpha_j)$ from the trusted party, outputs it, and never outputs $\perp$. Observe that none of the corrupted parties have input and so the adversary has no influence on the output of the honest parties. We begin by showing that this always holds in a *real execution* as well; i.e., in a real execution each honest party P_j always outputs $q(\alpha_j)$ and never outputs $\perp$.

Since the dealer is honest, it chooses a bivariate polynomial as described in the protocol and sends each party the prescribed values. In this case, an honest party P_j always outputs either $f_j(0) = S(0, \alpha_j) = q(\alpha_j)$ or $\perp$. This is due to the fact that its polynomial $f_j(x)$ will never be changed, because it can only be changed if a reveal$(j, f'_j(x), g_j(y))$ message is sent with $f'_j(x) \neq f_j(x)$. However, an honest dealer never does this. Thus, it remains to show that P_j never outputs $\perp$. In order to see this, recall that an honest party outputs $f_j(0)$ and not $\perp$ if and only if at least $n - t$ parties broadcast consistent. Thus, it suffices to show that all honest parties broadcast consistent. An honest party P_j broadcasts consistent if and only if the following conditions hold:

1. The dealer resolves all conflicts: Whenever a pair of complaint messages complaint(k, ℓ, u_1, v_1) and complaint(ℓ, k, u_2, v_2) were broadcast such that $u_1 \neq v_2$ and $v_1 \neq u_2$ for some k and ℓ, the dealer broadcasts a reveal message for ℓ or k or both in Step 4a (or else P_j would not broadcast consistent as specified in Step 5a).
2. The dealer did not broadcast reveal$(j, f_j(x), g_j(y))$. (See Step 5(b)i.)
3. Every revealed polynomial fits P_j's polynomials: Whenever the dealer broadcasts a message reveal$(k, f_k(x), g_k(y))$, it holds that $g_k(\alpha_j) = f_j(\alpha_k)$ and $f_k(\alpha_j) = g_j(\alpha_k)$. (See Step 5(b)ii.)

Since the dealer is honest, whenever there is a conflict between two parties, the dealer will broadcast a reveal message. This is due to the fact that if $u_1 \neq v_2$ or $u_2 \neq v_1$, it cannot hold that both (u_1, v_1) and (u_2, v_2) are consistent with $S(x, y)$ (i.e., it cannot be that $u_1 = S(\alpha_\ell, \alpha_k)$ and $v_1 = S(\alpha_k, \alpha_\ell)$ as well as $u_2 = S(\alpha_k, \alpha_\ell)$ and $v_2 = S(\alpha_\ell, \alpha_k)$). Thus, by its instructions, the dealer will broadcast at least one reveal message, and so condition (1) holds. In addition, it is immediate that since the dealer is honest, condition (3) also holds. Finally, the dealer broadcasts a reveal$(j, f_j(x), g_j(y))$ message if and only if P_j sends a complaint with an *incorrect* pair (u, v); i.e., P_j broadcast (j, k, u, v) where either $u \neq f_j(\alpha_k)$ or $v \neq g_j(\alpha_k)$. However, since both the dealer and P_j are honest, any complaint sent by P_j will be with the correct (u, v) values. Thus, the dealer will not broadcast a reveal of P_j's polynomials and condition (2) also holds. We conclude that every honest party broadcasts consistent and so all honest parties P_j output $f_j(0) = q(\alpha_j)$, as required.

Since the outputs of the honest parties are fully determined by the honest dealer's input, it remains to show the existence of an ideal model adversary/simulator $\mathcal{SIM}$ that can generate the *view of the adversary* $\mathcal{A}$ in an execution of the real protocol, given only the outputs $q(\alpha_i)$ of the corrupted parties P_i for every $i \in I$.

The Simulator $\mathcal{SIM}$

- $\mathcal{SIM}$ *invokes* $\mathcal{A}$ *on the auxiliary input* z.
- Interaction with the trusted party: $\mathcal{SIM}$ *receives the output values* $\{q(\alpha_i)\}_{i\in I}$.
- Generating the view of the corrupted parties: $\mathcal{SIM}$ *chooses any polynomial* $q'(x)$ *under the constraint that* $q'(\alpha_i) = q(\alpha_i)$ *for every* $i \in I$. *Then,* $\mathcal{SIM}$ *runs all honest parties* (*including the honest dealer*) *in an interaction with* $\mathcal{A}$, *with the dealer input polynomial as* $q'(x)$.
- $\mathcal{SIM}$ *outputs whatever* $\mathcal{A}$ *outputs, and halts.*

We now prove that the distribution generated by $\mathcal{SIM}$ is as required. First, observe that all that the corrupted parties see in the simulation by $\mathcal{SIM}$ is determined by the adversary and the *sequence* of polynomial pairs $\{(f_i(x), g_i(y))\}_{i\in I}$, where $f_i(x)$ and $g_i(y)$ are selected based on $q'(x)$, as described in the protocol. In order to see this, note that the only information sent after Round 1 are parties' complaints, complaint resolutions, and consistent messages. However, when the dealer is honest any complaint sent by an honest party P_j can only be due it receiving incorrect (u_i, v_i) from a corrupted party P_i (i.e., where either $u_i \neq f_j(\alpha_i)$ or $v_i \neq g_j(\alpha_i)$ or both). Such a complaint is of the form $(j, i, f_j(\alpha_i), g_j(\alpha_i))$, which equals $(j, i, g_i(\alpha_j), f_i(\alpha_j))$ since the dealer is honest, and so this complaint is determined by $(f_i(x), g_i(x))$ where $i \in I$. In addition, since the honest parties' complaints always contain correct values, the dealer can only send reveal messages reveal$(i, f_i(x), g_i(x))$ where $i \in I$; once again this information is already determined by the polynomial pairs of Round 1. Thus, all of the messages sent by $\mathcal{SIM}$ in the simulation can be computed from the sequence $\{(f_i(x), g_i(y))\}_{i\in I}$ only. Next, observe that the above is also true for a real protocol execution as well. Thus, the only difference between the real and ideal executions is whether the sequence $\{(f_i(x), g_i(y))\}_{i\in I}$ is based on the real polynomial $q(x)$ or the simulator-chosen polynomial $q'(x)$. However, by Claim 5.4 these distributions (i.e., $\{(f_i(x), g_i(y))\}_{i\in I}$) of are identical). This completes the proof of the case that the dealer is honest.

Case 2: The Dealer is Corrupted

In this case, the adversary $\mathcal{A}$ controls the dealer. Briefly speaking, the simulator $\mathcal{SIM}$ just plays the role of all honest parties. Recall that all actions of the parties, apart from the dealer, are deterministic and that these parties have no inputs. If the simulated execution is such that the parties output $\perp$, the simulator sends an invalid polynomial (say $q(x) = x^{2t}$) to the trusted party. Otherwise, the simulator uses the fact that it sees all "shares" sent by $\mathcal{A}$ to honest parties in order to interpolate and find the polynomial $q(x)$, which it then sends to the trusted party computing the functionality. That is, here the simulator invokes the trusted party after simulating an execution of the protocol. We now formally describe the simulator:

The Simulator $\mathcal{SIM}$

1. *$\mathcal{SIM}$ invokes $\mathcal{A}$ on its auxiliary input z.*
2. *$\mathcal{SIM}$ plays the role of all the $n - |I|$ honest parties interacting with $\mathcal{A}$, as specified by the protocol, running until the end.*
3. *Let* num *be the number of (honest and corrupted) parties P_j that broadcast* $\mathsf{consistent}$ *in the simulation*:
   (a) *If* $\mathsf{num} < n - t$, *then $\mathcal{SIM}$ sends the trusted party the polynomial $q'(x) = x^{2t}$ as the dealer input (this causes the trusted party to send $\perp$ as output to all parties in the ideal model).*
   (b) *If* $\mathsf{num} \geq n - t$, *then $\mathcal{SIM}$ defines a degree-t polynomial $q'(x)$ as follows. Let $K \subset [n] \setminus I$ be the set of all honest parties that broadcast* $\mathsf{consistent}$ *in the simulation. $\mathcal{SIM}$ finds the unique degree-t bivariate polynomial S that is guaranteed to exist by Claim 5.3 for this set K (later we will show why Claim 5.3 can be used). Then, $\mathcal{SIM}$ defines $q'(x) = S(0, x)$ and sends it to the trusted party (we stress that $q'(x)$ is not necessarily equal to the polynomial $q(x)$ that the dealer—equivalently P_1—receives as input).*
   $\mathcal{SIM}$ receives the output $\{q'(\alpha_i)\}_{i \in I}$ of the corrupted parties from the trusted party. (Since these values are already known to $\mathcal{SIM}$, they are not used. Nevertheless, $\mathcal{SIM}$ must send $q'(x)$ to the trusted party since this results in the honest parties receiving their output from F_{VSS}.)
4. *$\mathcal{SIM}$ halts and outputs whatever $\mathcal{A}$ outputs.*

Observe that all parties, as well as the simulator, are *deterministic* since the only party who tosses coins in the protocol is the honest dealer (where here the dealer is played by $\mathcal{A}$ and we can assume that $\mathcal{A}$ is deterministic because its auxiliary input can contain the "best" random coins for its attack). Thus, the outputs of all parties are fully determined both in the real execution of the protocol with $\mathcal{A}$ and in the ideal execution with $\mathcal{SIM}$. We therefore show that the outputs of the adversary and the parties in a real execution with $\mathcal{A}$ are equal to the outputs in an ideal execution with $\mathcal{SIM}$.

First, observe that the simulator plays the role of all the honest parties in an ideal execution, following the exact protocol specification. Since the honest parties have no input, the messages sent by the simulator in the ideal execution are exactly the same as those sent by the honest parties in a real execution of the protocol. Thus, the value that is output by $\mathcal{A}$ in a real execution *equals* the value that is output by $\mathcal{A}$ in the ideal execution with $\mathcal{SIM}$. It remains to show that the outputs of the honest parties are also the same in the real and ideal executions. Let OUTPUT_J denote the outputs of the parties P_j for all $j \in J$. We prove:

Claim 5.8. *Let $J = [n] \setminus I$ be the set of indices of the honest parties. For every adversary $\mathcal{A}$ controlling I including the dealer, every polynomial $q(x)$ and every auxiliary input $z \in \{0,1\}^*$ for $\mathcal{A}$, it holds that:*

$$\text{OUTPUT}_J\left(\text{REAL}_{\pi,\mathcal{A}(z),I}(q(x), \lambda, \ldots, \lambda)\right) = \text{OUTPUT}_J\left(\text{IDEAL}_{F_{VSS},\mathcal{S}(z),I}(q(x), \lambda, \ldots, \lambda)\right).$$

Proof. Let $\vec{x} = (q(x), \lambda, \ldots, \lambda)$ be the vector of inputs. We separately analyze the case that in the *real* execution some honest party outputs $\perp$ and the case where no honest party outputs $\perp$.

Case 1: There exists a $j \in J$ such that $\text{OUTPUT}_j(\text{REAL}_{\pi,\mathcal{A}(z),I}(q(x), \lambda, \ldots, \lambda)) = \perp$. We show that in this case all the honest parties output $\perp$ in both the real and ideal executions. Let j be such that $\text{OUTPUT}_j(\text{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x})) = \perp$. By the protocol specification, an honest party P_j outputs $\perp$ (in the real world) if and only if it receives less than $n - t$ "consistent" messages over the broadcast channel. Since these messages are broadcast, it holds that all the parties receive the same messages. Thus, if an honest P_j output $\perp$ in the real execution, then each honest party received less than $n-t$ such "consistent" messages, and so every honest party outputs $\perp$ (in the real execution).

We now claim that in the ideal execution, all honest parties also output $\perp$. The output of the honest parties in the ideal execution is determined by the trusted third party, based on the input sent by $\mathcal{SIM}$. It follows by the specification of $\mathcal{SIM}$ that all honest parties output $\perp$ if and only if $\mathcal{SIM}$ sends x^{2t} to the trusted third party. As we have mentioned, the simulator $\mathcal{SIM}$ follows the instructions of the honest parties exactly in the simulation. Thus, if in a real execution with $\mathcal{A}$ less than $n - t$ parties broadcast consistent, then the same is also true in the simulation with $\mathcal{SIM}$. (We stress that *exactly the same messages* are sent by $\mathcal{A}$ and the honest parties in a real protocol execution and in the simulation with $\mathcal{SIM}$.) Now, by the instructions of $\mathcal{SIM}$, if less than $n-t$ parties broadcast consistent, then num $< n - t$, and $\mathcal{SIM}$ sends $q(x) = x^{2t}$ to the trusted party. We conclude that all honest parties output $\perp$ in the ideal execution as well.

Case 2: For every $j \in J$ it holds that $\text{OUTPUT}_j(\text{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x})) \neq \perp$. By what we have discussed above, this implies that in the simulation with $\mathcal{SIM}$, at least $n - t$ parties broadcast consistent. Since $n \geq 3t + 1$ this implies that at least $3t + 1 - t \geq 2t + 1$ parties broadcast consistent. Furthermore, since there are at most t corrupted parties, we have that at least $t + 1$ *honest* parties broadcast consistent.

Recall that an honest party P_j broadcasts consistent if and only if the following conditions hold (cf. the case of honest dealer):

1. The dealer resolves all conflicts (Step 5a of the protocol).
2. The dealer did not broadcast reveal$(j, f_j(x), g_j(y))$ (Step 5(b)i of the protocol).
3. Every revealed polynomial fits P_j's polynomials (Step 5(b)ii of the protocol).

Let $K \subset [n]$ be the set of honest parties that broadcast consistent as in Step 3b of $\mathcal{SIM}$. For each of these parties the above conditions hold. Thus, for every $i, j \in K$ it holds that $f_i(\alpha_j) = g_j(\alpha_i)$ and so Claim 5.3 can be applied. This implies that there exists a *unique* bivariate polynomial S such that $S(x, \alpha_k) = f_k(x)$ and $S(\alpha_k, y) = g_k(y)$ for every $k \in K$. Since S is unique, it also defines a unique polynomial $q'(x) = S(0, x)$. Now, since $\mathcal{SIM}$ sends $q'(x)$ to the trusted party in an ideal execution, we have that all honest parties P_j output $q'(\alpha_j)$ in an ideal execution. We now prove that the same also holds in a real protocol execution.

We stress that the polynomial $q'(x)$ is defined as a deterministic function of the transcript of messages sent by $\mathcal{A}$ in a real or ideal execution. Furthermore, since the execution is deterministic, the exact same polynomial $q'(x)$ is defined in both the real and ideal executions. It therefore remains to show that each honest party P_j outputs $q'(\alpha_j)$ in a real execution. We first observe that any honest party P_k for $k \in K$ clearly outputs $q'(\alpha_k)$. This follows from the fact that by the protocol description, each party P_i that does not output $\perp$ outputs $f_i(0)$. Thus, each such P_k outputs $f_k(0)$. We have already seen that $q'(x)$ is the unique polynomial that passes through the points $(\alpha_k, f_k(0))$ and thus $q'(\alpha_k) = f_k(0)$ for every $k \in K$.

It remains to show that every honest party P_j for $j \notin K$ also outputs $q'(\alpha_j)$; i.e., it remains to show that every honest party P_j who did *not* broadcast consistent also outputs $q'(\alpha_j)$. Let $f'_j(x)$ and $g'_j(x)$ be the polynomials that P_j holds after the possible replacement in Step 5(b)i of the protocol (note that these polynomials may be different from the original polynomials that P_j received from the dealer at the first stage). We stress that this party P_j did not broadcast consistent, and therefore we cannot rely on the conditions above. However, for every party P_k ($k \in K$) who broadcast consistent, we are guaranteed that the polynomials $f_k(x)$ and $g_k(y)$ are consistent with the values of the polynomials of P_j; that is, it holds that $f_k(\alpha_j) = g'_j(\alpha_k)$ and $g_k(\alpha_j) = f'_j(\alpha_k)$. This follows from the fact that all conflicts are properly resolved (and so if they were inconsistent then a reveal message must have been sent to make them consistent). This implies that for $t+1$ points $k \in K$, it holds that $f'_j(\alpha_k) = S(\alpha_k, \alpha_j)$, and so since $f'_j(x)$ is a polynomial of degree t (by the truncation instruction; see the protocol specification) it follows that $f'_j(x) = S(x, \alpha_j)$ (because both are degree-t polynomials in x). Thus, $f'_j(0) = S(0, \alpha_j)$ and we have that P_j outputs $S(0, \alpha_j)$. This completes the proof because $q'(\alpha_j) = S(0, \alpha_j)$, as described above. □

This completes the proof of Theorem 5.7. □

Efficiency We remark that in the case that no parties behave maliciously in Protocol 5.6, the protocol merely involves the dealer sending two polynomials to each party, and each party sending two field elements to every other party. Specifically, if no party broadcasts a complaint, then the protocol can conclude immediately after Round 3.

6. Multiplication in the Presence of Malicious Adversaries

6.1. *High-Level Overview*

In this section, we show how to securely compute shares of the product of shared values, in the presence of a malicious adversary controlling any $t < n/3$ parties. We use the simplification of the original multiplication protocol of Ben-Or et al. [7] that appears in [19]. We start with a short overview of the simplification of Gennaro et al. [19] in the semi-honest model, and then we show how to move to the malicious case.

Assume that the values on the input wires are a and b, respectively, and that each party holds degree-t shares a_i and b_i. Recall that the values $a_i \cdot b_i$ define a (non random) degree-$2t$ polynomial that hides $a \cdot b$. The semi-honest multiplication protocol of Ben-Or et al. [7] works by first rerandomizing this degree-$2t$ polynomial, and then reducing its degree to degree-t while preserving the constant term which equals $a \cdot b$ (see Sect. 4.3). Recall also that the degree-reduction works by running the BGW protocol for a linear function, where the first step involves each party sharing its input by a degree-t polynomial. In our case, the parties' inputs are themselves shares of a degree-$2t$ polynomial, and thus each party "subshares" its share.

The method of Gennaro et al. [19] simplifies this protocol by replacing the two different stages of rerandomization and degree reduction with a single step. The simplification is based on an observation that a specific linear combination of all the subshares of all $a_i \cdot b_i$ defines a *random* degree-t polynomial that hides $a \cdot b$ (where the randomness of the polynomial is derived from the randomness of the polynomials used to define the subshares). Thus, the protocol involves first subsharing the share product values $a_i \cdot b_i$, and then carrying out a local linear combination of the obtained subshares.

The main problem and difficulty that arises in the case of malicious adversaries is that corrupted parties may not subshare the correct values $a_i \cdot b_i$. We therefore need a mechanism that forces the corrupted parties to distribute the correct values, without revealing any information. Unfortunately, it is not possible to simply have the parties VSS subshare their share products $a_i \cdot b_i$ and then use error correction to correct any corrupt values. This is due to the fact that the shares $a_i \cdot b_i$ lie on a degree-$2t$ polynomial, which in turn defines a Reed–Solomon code of parameters $[n, 2t+1, n-2t]$. For such a code, it is possible to correct up to $\frac{n-2t-1}{2}$ errors (see Sect. 5.2); plugging in $n = 3t + 1$ we have that it is possible to correct up to $\frac{t}{2}$ errors. However, there are t corrupted parties and so incorrect values supplied by more than half of them cannot be corrected.[5] The BGW protocol therefore forces the parties to distribute correct values, using the following steps:

1. The parties first distribute subshares of their input shares on each wire (rather than the subshares of the product of their input shares) to all other parties in a verifiable way. That is, each party P_i distributes subshares of a_i and subshares of b_i. Observe that the input shares are points on degree-t polynomials. Thus, these shares constitute a Reed–Solomon code with parameters $[n, t+1, n-t]$ for which it is possible to correct up to t errors. There is therefore enough redundancy to correct errors, and so any incorrect values provided by corrupted parties can be corrected. This operation is carried out using the $F_{VSS}^{subshare}$ functionality, described in Sect. 6.4.
2. Next, each party distributes subshares of the product $a_i \cdot b_i$. The protocol for subsharing the product uses the separate subshares of a_i and b_i obtained in the previous step, in order to verify that the correct product $a_i \cdot b_i$ is shared. Stated differently, this step involves a protocol for verifying that a party distributes shares of $a_i \cdot b_i$ (via a degree-t polynomial), given shares of a_i and shares of b_i (via degree-t polynomials). This step is carried out using the F_{VSS}^{mult} functionality, described in Sect. 6.6. In order to implement this step, we introduce a new functionality called F_{eval} in Sect. 6.5.

[5] We remark that in the case of $t < n/4$ (i.e., $n \geq 4t + 1$), the parties can correct errors directly on degree-$2t$ polynomials. Therefore, the parties can distribute subshares of the products $a_i \cdot b_i$, and correct errors on these shares using (a variant of) the $F_{VSS}^{subshare}$ functionality directly. Thus, overall, the case of $t < n/4$ is significantly simpler, since there is no need for the F_{VSS}^{mult} subprotocol that was mentioned in the second step described above. A full specification of this simplification is described in "Appendix"; the description assumes familiarity with the material appearing in Sects. 6.2, 6.3, 6.4 and 6.7, and therefore should be read after these sections.

3. Finally, after the previous step, all parties verifiably hold (degree-t) subshares of all the products $a_i \cdot b_i$ of every party. As described above, shares of the product $a \cdot b$ can be obtained by computing a linear function of the subshares obtained in the previous step. Thus, each party just needs to carry out a local computation on the subshares obtained. This is described in Sect. 6.7.

Before we show how to securely compute the $F_{VSS}^{subshare}$ functionality, we present relevant preliminaries in Sects. 6.2 and 6.3. Specifically, in Sect. 6.2 we introduce the notion of *corruption-aware functionalities*. These are functionalities whose behavior may depend on which parties are corrupted. We use this extension of standard functionalities in order to prove the BGW protocol in a modular fashion. Next, in Sect. 6.3 we present a subprotocol for securely computing matrix multiplication over a shared vector. This will be used in the protocol for securely computing $F_{VSS}^{subshare}$, which appears in Sect. 6.4.

6.2. *Corruption-Aware Functionalities and Their Use*

In the standard definition of secure computation (see Sect. 2.2 and [8,20]) the functionality defines the desired input/output behavior of the computation. As such, it merely receives inputs from the parties and provides outputs. However, in some cases, we wish to provide the corrupted parties, equivalently the adversary, with some additional power over the honest parties.

In order to see why we wish to do this, consider the input-sharing phase of the BGW protocol, where each party distributes its input using secret sharing. This is achieved by running n executions of VSS where in the ith copy party P_i plays the dealer with a polynomial $q_i(x)$ defining its input. The question that arises now is what security is obtained when running these VSS invocations in *parallel*, and in particular we need to define *the ideal functionality that such parallel VSS executions fulfills*. Intuitively, the security of the VSS protocol guarantees that all shared values are independent. Thus, one could attempt to define the "parallel VSS" functionality as follows:

FUNCTIONALITY 6.1 (Parallel VSS (naive attempt)—F_{VSS}^n).

1. The parallel F_{VSS}^n functionality receives inputs $q_1(x), \dots, q_n(x)$ from parties $P_1, \dots, P_n$, respectively. If P_i did not send a polynomial $q_i(x)$, or $\deg(q_i) > t$, then F_{VSS}^n defines $q_i(x) = \perp$ for every x.
2. For every $i = 1, \dots, n$, the functionality F_{VSS}^n sends $(q_1(\alpha_i), \dots, q_n(\alpha_i))$ to party P_i.

This is the naive extension of the single F_{VSS} functionality (Functionality 5.5), and at first sight seems to be the appropriate ideal functionality for a protocol consisting of *n parallel* executions of Protocol 5.6 for computing F_{VSS}. However, we now show that this protocol does not securely compute the parallel VSS functionality as defined.

Recall that the adversary is *rushing*, which means that it can receive the honest parties' messages in a given round before sending its own. In this specific setting, the adversary can see the corrupted parties' shares of the honest parties' polynomials before it chooses the corrupted parties' input polynomials (since these shares of the honest parties' polynomials are all sent to the corrupted parties in the first round of Protocol 5.6). Thus, the adversary can choose the corrupted parties' polynomials in a way that is related to the honest parties' polynomials. To be specific, let P_j be an honest party with input $q_j(x)$, and let P_i be a corrupted party. Then, the adversary can first see P_i's share $q_j(\alpha_i)$, and then choose $q_i(x)$ so that $q_i(\alpha_i) = q_j(\alpha_i)$, for example. In contrast, the adversary in the ideal model with F_{VSS}^n *cannot* achieve this effect since it receives no information about the honest parties' polynomials before all input polynomials, including those of the corrupted parties, are sent to the trusted party. Thus, n parallel executions of Protocol 5.6 does *not* securely compute F_{VSS}^n as defined in Functionality 6.1.

Despite the above, we stress that in many cases (and, in particular, in the application of parallel VSS in the BGW protocol) this adversarial capability is of no real concern. Intuitively, this is due to the fact that $q_j(\alpha_i)$ is actually independent of the constant term $q_j(0)$ and so making $q_i(\alpha_i)$ depend on $q_j(\alpha_i)$ is of no consequence in this application. Nevertheless, the adversary *can* set $q_i(x)$ in this way in the real protocol (due to rushing), but *cannot do so* in the ideal model with functionality F_{VSS}^n (as in Functionality 6.1). Therefore, the protocol consisting of n parallel calls to F_{VSS} does *not* securely compute the F_{VSS}^n functionality. Thus, one has to either modify the protocol or change the functionality definition, or both. Observe that the fact that in some applications we don't care about this adversarial capability is immaterial: The problem is that the protocol does not securely compute Functionality 6.1 and thus something has to be changed.

One possible modification to both the protocol and functionality is to run the F_{VSS} executions sequentially in the real protocol and define an ideal (reactive) functionality where each party P_i first receives its shares $q_1(\alpha_i), \dots, q_{i-1}(\alpha_i)$ from the previous VSS invocations before sending its own input polynomial $q_i(x)$. This solves the aforementioned problem since the ideal (reactive) functionality allows each party to make its polynomial depend on shares previously received. However, this results in a protocol that is not constant round, which is a significant disadvantage.

Another possible modification is to leave the protocol unmodified (with n parallel calls to F_{VSS}), and change the ideal functionality as follows. First, the *honest* parties send their input polynomials $q_j(x)$ (for every $j \notin I$). Next, the corrupted parties receive their shares on these polynomials (i.e., $q_j(\alpha_i)$ for every $j \notin I$ and $i \in I$), and finally the corrupted parties send their polynomials $q_i(x)$ (for every $i \in I$) to the trusted party. This reactive functionality captures the capability of the adversary to choose the corrupted parties' polynomials based on the shares $q_j(\alpha_i)$ that it views on the honest parties' polynomials, but nothing more. Formally, we define:

This modification to the definition of F_{VSS}^n solves our problem. However, the standard definition of security, as referred in Sect. 2.2, does not allow us to define a functionality in this way. This is due to the fact that the standard formalism does not distinguish between honest and malicious parties. Rather, the functionality is supposed to receive inputs from each honest and corrupt party in the same way, and in particular does not "know" which parties are corrupted. We therefore augment the standard formalism to allow corruption-aware functionalities (CA functionalities) that *receive the set I of the identities of the corrupted parties as additional auxiliary input* when invoked. We proceed by describing the changes required to the standard (stand-alone) definition of security of Sect. 2.2 in order to incorporate corruption awareness.

FUNCTIONALITY 6.2 (Corruption-aware parallel VSS—F^n_{VSS}).
F^n_{VSS} receives a set of indices $I \subseteq [n]$ and works as follows:

1. F^n_{VSS} receives an input polynomial $q_j(x)$ from every *honest* P_j ($j \notin I$).
2. F^n_{VSS} sends the (ideal model) adversary the corrupted parties' shares $\{q_j(\alpha_i)\}_{j \notin I}$ for every $i \in I$, based on the honest parties' polynomials.
3. F^n_{VSS} receives from the (ideal model) adversary an input polynomial $q_i(x)$ for every $i \in I$.
4. F^n_{VSS} sends the shares $(q_1(\alpha_j), \ldots, q_n(\alpha_j))$ to every party P_j ($j = 1, \ldots, n$). If $\deg(q_i(x)) > t$ then $\perp$ is sent in place of $q_i(\alpha_j)$.[6]

Definition The formal definition of security for a corruption-aware functionality is the same as Definition 2.3 with the sole change being that f is a function of the subset of corrupted parties and the inputs; formally, $f\colon 2^{[n]} \times \left(\{0,1\}^*\right)^n \to \left(\{0,1\}^*\right)^n$. We denote by $f_I(\vec{x}) = f(I, \vec{x})$ the function f with the set of corrupted parties fixed to $I \subset [n]$. Then, we require that for every subset I (of cardinality at most t), the distribution $\text{IDEAL}_{f_I,\mathcal{S}(z),I}(\vec{x})$ is distributed identically to $\text{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x})$. We stress that in the ideal model, the subset I that is given to a corruption-aware functionality as auxiliary input (upon initialization) is the same subset I of corrupted parties that the adversary controls. Moreover, the functionality receives this subset I at the very start of the ideal process, in the exact same way as the (ideal model) adversary receives the auxiliary input z, the honest parties receive their inputs, and so on. We also stress that the honest parties (both in the ideal and real models) do *not* receive the set I, since this is something that is of course not known in reality (and so the security notion would be nonsensical). Formally,

Definition 6.3. Let $f\colon 2^{[n]} \times \left(\{0,1\}^*\right)^n \to \left(\{0,1\}^*\right)^n$ be a *corruption-aware* n-ary functionality and let π be a protocol. We say that π is t-secure for f if for every probabilistic adversary $\mathcal{A}$ in the real model, there exists a probabilistic adversary $\mathcal{S}$ of comparable complexity in the ideal model, such that for every $I \subset [n]$ of cardinality at most t, every $\vec{x} \in (\{0,1\}^*)^n$ where $|x_1| = \cdots = |x_n|$, and every $z \in \{0,1\}^*$, it holds that: $\{\text{IDEAL}_{f_I,\mathcal{S}(z),I}(\vec{x})\} \equiv \{\text{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x})\}$.

We stress that since we only consider static adversaries here, the set I is fully determined before the execution begins, and thus this is well defined.

This idea of having the behavior of the functionality depend on the adversary and/or the identities of the corrupted parties was introduced by Canetti [9] in order to provide more flexibility in defining functionalities, and is heavily used in the universal composability framework.[7]

[6] It actually suffices to send the shares $(q_1(\alpha_j), \ldots, q_n(\alpha_j))$ only to parties P_j for $j \notin I$ since all other parties have already received these values. Nevertheless, we present it in this way for the sake of clarity.

[7] In the UC framework, the adversary can communicate directly with the ideal functionality and it is mandated that the adversary notifies the ideal functionality (i.e., trusted party) of the identities of all corrupted parties. Furthermore, ideal functionalities often utilize this information (i.e., they are corruption aware) since the way that the universal composability framework is defined typically requires functionalities to treat the inputs of honest and corrupted parties differently. See Section 6 of the full version of [9] for details.

The Hybrid Model and Modular Composition In the hybrid model, where the parties have oracle tapes for some ideal functionality (trusted party), in addition to regular communication tapes, the same convention for corruption awareness is followed as in the ideal model. Specifically, an execution in the $\mathcal{G}_I$-hybrid model, denoted $\text{HYBRID}^{\mathcal{G}_I}_{f,\mathcal{A}(z),I}(\vec{x})$, is parameterized by the set I of corrupted parties, and this set I is given to functionality $\mathcal{G}$ upon initialization of the system just like the auxiliary input is given to the adversary. As mentioned above, I is fixed ahead of time and so this is well defined. We stress again that the honest parties do not know the set of indices I, and real messages sent by honest parties and their input to the ideal functionality are independent of I.

In more detail, in an ideal execution the behavior of the trusted party depends heavily on the set of corrupted parties I, and in some sense, its exact code is fixed *only after* we determine the set of corrupted parties I. In contrast, in a real execution the specification of the protocol is independent of the set I, and the code that the honest parties execute is fixed ahead of time and is the same one *for any* set of corrupted parties I. An execution in the hybrid model is something in between: the code of the honest parties is independent of I and is fixed ahead of time (like in the real model); however, the code of the aiding functionality is fixed only after we set I (as in the ideal model).

Throughout our proof of security of the BGW protocol for malicious adversaries, some of the functionalities we use are corruption aware and some are not; in particular, as we will describe, our final functionality for secure computation with the BGW protocol is *not* corruption aware. In order to be consistent with respect to the definition, we work with corruption-aware functionalities only and remark that *any* ordinary functionality f (that is *not* corruption aware) can be rewritten as a fictitiously corruption-aware functionality f_I where the functionality just ignores the auxiliary input I. An important observation is that a protocol that securely computes this fictitiously corruption-aware functionality, securely computes the original functionality in the standard model (i.e., when the functionality does not receive the set I as an auxiliary input). This holds also for protocols that use corruption-aware functionalities as subprotocols (as we will see, this is the case with the final BGW protocol). This observation relies on the fact that a protocol is always corruption unaware, and that the simulator knows the set I in *both* the corruption aware and the standard models. Thus, the simulator is able to simulate the corruption-aware subprotocol, even in the standard model. Indeed, since the corruption-aware functionality f_I ignores the set I, and since the simulator knows I in both models, the two ensembles $\text{IDEAL}_{f_I,\mathcal{S}(z),I}(\vec{x})$ (in the corruption-aware model) and $\text{IDEAL}_{f,\mathcal{S}(z),I}(\vec{x})$ (in the standard model) are identical. Due to this observation, we are able to conclude that the resulting BGW protocol securely computes any standard (not corruption-aware) functionality in the *standard model*, even though it uses corruption-aware subprotocols.

Regarding composition, the sequential modular composition theorems of [8, 20] do not consider corruption-aware functionalities. Nevertheless, it is straightforward to see that the proofs hold also for this case, with no change whatsoever. Thus, the method described in Sect. 2.3 for proving security in a modular way can be used with corruption-aware functionalities as well.

Discussion The augmentation of the standard definition with corruption-aware functionalities enables more flexibility in protocol design. Specifically, it is possible to model the situation where corrupted parties can learn more than just the specified output, or can obtain some other "preferential treatment" (like in the case of parallel VSS where they are able to set their input polynomials as a partial function of the honest parties' input). In some sense, this implies a weaker security guarantee than in the case where all parties (honest and corrupted) receive the same treatment. However, since the ideal functionality is specified so that the "weakness" is explicitly stated, the adversary's advantage is well defined.

This approach is not foreign to modern cryptography and has been used before. For example, secure encryption is defined while allowing the adversary a negligible probability of learning information about the plaintext. A more significant example is the case of two-party secure computation. In this case, the ideal model is defined so that the corrupted party explicitly receives the output first and can then decide whether or not the honest party also receives output. This is weaker than an ideal model in which both parties receive output and so "complete fairness" is guaranteed. However, since complete fairness cannot be achieved (in general) without an honest majority, this weaker ideal model is used, and the security weakness is explicitly modeled.

In the context of this paper, we use corruption awareness in order to enable a modular analysis of the BGW protocol. In particular, for some of the subprotocols used in the BGW protocol, it seems hard to define an appropriate ideal functionality that is not corruption aware. Nevertheless, our final result regarding the BGW protocol is for standard functionalities. That is, when we state that *every functionality* can be securely computed by BGW (with the appropriate corruption threshold), we refer to regular functionalities and not to corruption-aware ones.

The reason why the final BGW protocol works for corruption unaware functionalities *only* is due to the fact that the protocol emulates the computation of a *circuit* that computes the desired functionality. However, not every corruption-aware functionality can be computed by a circuit that receives inputs from the parties only, without also having the identities of the set of corrupted parties as auxiliary input. Since the real protocol is never allowed to be "corruption aware," this means that such functionalities cannot be realized by the BGW protocol. We remark that this is in fact *inherent*, and there exist corruption-aware functionalities that cannot be securely computed by *any* protocol. In particular, consider the functionality that just announces to all parties who is corrupted. Since a corrupted party may behave like an honest one, it is impossible to securely compute such a functionality.

Finally, we note that since we already use corruption awareness anyhow in our definitions of functionalities (for the sake of feasibility and/or efficiency), we sometimes also use it in order to simplify the definition of a functionality. For example, consider a secret-sharing reconstruction functionality. As we have described in Sect. 5.2, when $t < n/3$, it is possible to use Reed–Solomon error correction to reconstruct the secret, even when up to t incorrect shares are received. Thus, an *ideal* functionality for reconstruction can be formally defined by having the trusted party run the Reed–Solomon error correction procedure. Alternatively, we can define the ideal functionality so that it receive shares from the *honest parties* only, and reconstructs the secret based on these shares only (which are guaranteed to be correct). This latter formulation is corruption aware, and has the advantage of making it clear that the adversary cannot influence the outcome of the reconstruction in any way.

Convention For the sake of clarity, we will describe (corruption-aware) functionalities as having direct communication with the (ideal) adversary. In particular, the corrupted parties will not send input or receive output, and all such communication will be between the adversary and functionality. This is equivalent to having the corrupted parties send input as specified by the adversary.

Moreover, we usually omit the set of corrupted parties I in the notation of a corruption-aware functionality (i.e., we write $\mathcal{G}$ instead of $\mathcal{G}_I$). However, in the definition of any corruption-aware functionality, we add an explicit note that the functionality receives as auxiliary input the set of corrupted parties I. In addition, for any protocol in the corruption-aware hybrid model, we add an "aiding ideal functionality initialization" step, to explicitly emphasize that the aiding ideal functionalities receive the set I upon initialization.

6.3. *Matrix Multiplication in the Presence of Malicious Adversaries*

We begin by showing how to securely compute the matrix multiplication functionality, that maps the input vector $\vec{x}$ to $\vec{x} \cdot A$ for a fixed matrix A, where the ith party holds x_i and all parties receive the entire vector $\vec{x} \cdot A$ as output. Beyond being of interest in its own right, this serves as a good warm-up to secure computation in the malicious setting. In addition, we will explicitly use this as a subprotocol in the computation of $F_{VSS}^{subshare}$ in Sect. 6.4.

The basic matrix multiplication functionality is defined by a matrix $A \in \mathbb{F}^{n \times m}$, and the aim of the parties is to securely compute the length-m vector $(y_1, \ldots, y_m) = (x_1, \ldots, x_n) \cdot A$, where $x_1, \ldots, x_n \in \mathbb{F}$ are their respective inputs. (Indeed, the case $m = 1$ is also of interest, but we shall need $m = 2t$.) We will actually need to define something more involved, but we begin by explaining how one can securely compute the basic functionality. Note first that matrix multiplication is a linear functionality (i.e., it can be computed by circuits containing only addition and multiplication-by-constant gates). Thus, we can use the same methodology as was described at the end of Sect. 4.2 for privately computing any linear functionality, in the semi-honest model. Specifically, the inputs are first shared. Next, each party locally computes the linear functionality on the shares it received. Finally, the parties send their resulting shares in order to reconstruct the output. The difference here in the malicious setting is simply that the *verifiable* secret-sharing functionality is used for sharing the inputs, and Reed–Solomon decoding (as described in Sect. 5.2) is used for reconstructing the output. Thus, the basic matrix multiplication functionality can be securely computed as follows:

1. *Input-sharing phase* Each party P_i chooses a random polynomial $g_i(x)$ under the constraint that $g_i(0) = x_i$. Then, P_i shares its polynomial $g_i(x)$ using the ideal F_{VSS} functionality. After all polynomials are shared, party P_i has the shares $g_1(\alpha_i), \ldots, g_n(\alpha_i)$.
2. *Matrix multiplication emulation phase* Given the shares from the previous step, each party computes its Shamir-share of the output vector of the matrix multiplication by computing $\vec{y}^i = (g_1(\alpha_i), \ldots, g_n(\alpha_i)) \cdot A$. Note that:

$$\vec{y}^i = (g_1(\alpha_i), \ldots, g_n(\alpha_i)) \cdot A = [g_1(\alpha_i), g_2(\alpha_i), \ldots, g_n(\alpha_i)] \cdot \begin{bmatrix} a_{1,1} & \ldots & a_{1,m} \\ a_{2,1} & \ldots & a_{2,m} \\ \vdots & & \vdots \\ a_{n,1} & \ldots & a_{n,m} \end{bmatrix}$$

and so the jth element in $\vec{y}^i$ equals $\sum_{\ell=1}^{n} g_\ell(\alpha_i) \cdot a_{\ell,j}$. Denoting the jth element in $\vec{y}^i$ by y_j^i, we have that $y_j^1, \ldots, y_j^n$ are Shamir-shares of the jth element of $\vec{y} = (g_1(0), \ldots, g_n(0)) \cdot A$.

3. *Output reconstruction phase*:
   (a) Each party P_i sends its vector $\vec{y}^i$ to all other parties.
   (b) Each party P_i reconstructs the secrets from all the shares received, thereby obtaining $\vec{y} = (g_1(0), \ldots, g_n(0)) \cdot A$. This step involves running (local) error correction on the shares, in order to neutralize any incorrect shares sent by the malicious parties. Observe that the vectors sent in the protocol constitute the rows in the matrix

$$\begin{bmatrix} \leftarrow \vec{y}^1 \rightarrow \\ \leftarrow \vec{y}^2 \rightarrow \\ \vdots \\ \leftarrow \vec{y}^n \rightarrow \end{bmatrix} = \begin{bmatrix} \sum_{\ell=1}^{n} g_\ell(\alpha_1) \cdot a_{\ell,1} & \cdots & \sum_{\ell=1}^{n} g_\ell(\alpha_1) \cdot a_{\ell,m} \\ \sum_{\ell=1}^{n} g_\ell(\alpha_2) \cdot a_{\ell,1} & \cdots & \sum_{\ell=1}^{n} g_\ell(\alpha_2) \cdot a_{\ell,m} \\ \vdots & & \vdots \\ \sum_{\ell=1}^{n} g_\ell(\alpha_n) \cdot a_{\ell,1} & \cdots & \sum_{\ell=1}^{n} g_\ell(\alpha_n) \cdot a_{\ell,m} \end{bmatrix}$$

and the jth *column* of the matrix constitutes Shamir-shares on the polynomial with constant term $\sum_{\ell=1}^{n} g_\ell(0) \cdot a_{j,\ell}$, which is the jth element in the output. Thus, Reed–Solomon error correction can be applied to the columns in order to correct any incorrect shares and obtain the correct output.

The above protocol computes the correct output: The use of F_{VSS} in the first step prevents any malicious corrupted party from sharing an invalid polynomial, while the use of error correction in the last step ensures that the corrupted parties cannot adversely influence the output.

However, as we have mentioned, we need matrix multiplication in order to secure compute the $F_{VSS}^{subshare}$ functionality in Sect. 6.4. In this case, the functionality that is needed is a little more involved than basic matrix multiplication. First, instead of each party P_i inputting a value x_i, we need its input to be a degree-t polynomial $g_i(x)$ and the constant term $g_i(0)$ takes the place of x_i.[8] Next, in addition to obtaining the result $\vec{y} = (g_1(0), \ldots, g_n(0)) \cdot A$ of the matrix multiplication, each party P_i also outputs the shares $g_1(\alpha_i), \ldots, g_n(\alpha_i)$ that it received on the input polynomials of the parties. Based on the above, one could define the functionality as

$$F_{mat}^{A}(g_1, \ldots, g_n) = \left(\left(\vec{y}, \{g_\ell(\alpha_1)\}_{\ell=1}^{n}\right), \left(\vec{y}, \{g_\ell(\alpha_2)\}_{\ell=1}^{n}\right) \ldots, \left(\vec{y}, \{g_\ell(\alpha_n)\}_{\ell=1}^{n}\right) \right),$$

[8]This is needed because in $F_{VSS}^{subshare}$ the parties need to output $g_i(x)$ and so need to know it. It would be possible to have the functionality choose $g_i(x)$ and provide it in the output, but then exactly the same issue would arise. This is explained in more detail in the next paragraph.

where $\vec{y} = (g_1(0), \dots, g_n(0)) \cdot A$. Although this looks like a very minor difference, as we shall see below, it significantly complicates things. In particular, we will need to define a corruption-aware variant of this functionality.

We now explain why inputting polynomials $g_1(x), \dots, g_n(x)$ rather than values $x_1, \dots, x_n$ (and likewise outputting the shares) makes a difference. In the protocol that we described above for matrix multiplication, each party P_i sends its shares $\vec{y}^i$ of the output. Now, the vectors $\vec{y}^1, \dots, \vec{y}^n$ are *fully determined* by the input polynomials $g_1(x), \dots, g_n(x)$. However, in the ideal execution, the simulator only receives a subset of the shares and cannot simulate all of them. (Note that the simulator cannot generate random shares since the $\vec{y}^i$ vectors are fully determined by the input.) To be concrete, consider the case that only party P_1 is corrupted. In this case, the ideal adversary receives as output $\vec{y} = (g_1(0), \dots, g_n(0)) \cdot A$ and the shares $g_1(\alpha_1), \dots, g_n(\alpha_1)$. In contrast, the real adversary sees all of the vectors $\vec{y}^2, \dots, \vec{y}^n$ sent by the honest parties in the protocol. However, these vectors (or messages) are a *deterministic* function of the input polynomials $g_1(x), \dots, g_n(x)$ and of the fixed matrix A. Thus, the simulator in the ideal model must be able to generate the *exact* messages sent by the honest parties (recall that the distinguisher knows all of the inputs and outputs and so can verify that the output transcript is truly consistent with the inputs). But, it is impossible for a simulator who is given only $\vec{y}$ and the shares $g_1(\alpha_1), \dots, g_n(\alpha_1)$ to generate these exact messages, since it doesn't have enough information. In an extreme example, consider the case that $m = n$, the matrix A is the identity matrix, and the honest parties' polynomials are random. In this case, $\vec{y}^i = (g_1(\alpha_i), \dots, g_n(\alpha_i))$. By the properties of random polynomials, the simulator cannot generate $\vec{y}^i$ for $i \neq 1$ given only $\vec{y} = (g_1(0), \dots, g_n(0))$, the shares $(g_1(\alpha_1), \dots, g_n(\alpha_1))$ and the polynomial $g_1(x)$.

One solution to the above is to modify the protocol by somehow adding randomness, thereby making the $\vec{y}^i$ vectors not a deterministic function of the inputs. However, this would add complexity to the protocol and turns out to be unnecessary. Specifically, we only construct this protocol for its use in securely computing $F_{VSS}^{subshare}$, and the security of the protocol for computing $F_{VSS}^{subshare}$ is maintained even if the adversary explicitly learns the vector of m polynomials $\vec{Y}(x) = (Y_1(x), \dots, Y_m(x)) = (g_1(x), \dots, g_n(x)) \cdot A$. (Denoting the jth column of A by $(a_{1,j}, \dots, a_{n,j})^T$, we have that $Y_j(x) = \sum_{\ell=1}^{n} g_\ell(x) \cdot a_{\ell,j}$.) We therefore modify the functionality definition so that the adversary receives $\vec{Y}(x)$, thereby making it corruption aware (observe that the basic output $(g_1(0), \dots, g_n(0)) \cdot A$ is given by $\vec{Y}(0)$). Importantly, given this additional information, it is possible to simulate the protocol based on the methodology described above (VSS sharing, local computation, and Reed–Solomon reconstruction), and prove its security.

Before formally defining the F_{mat}^A functionality, we remark that we also use corruption awareness in order to deal with the fact that the first step of the protocol for computing F_{mat}^A involves running parallel VSS invocations, one for each party to distribute shares of its input polynomial. As we described in Sect. 6.2 this enables the adversary to choose the corrupted parties' polynomials $g_i(x)$ (for $i \in I$) after seeing the corrupted parties' shares on the honest parties' polynomials (i.e., $g_j(\alpha_i)$ for every $j \notin I$ and $i \in I$). We therefore model this capability in the functionality definition.

FUNCTIONALITY 6.4 (Functionality F^A_{mat} for matrix multiplication, with $A \in \mathbb{F}^{n\times m}$).
The F^A_{mat}-functionality receives as input a set of indices $I \subseteq [n]$ and works as follows:

1. F^A_{mat} receives the inputs of the honest parties $\{g_j(x)\}_{j\notin I}$; if a polynomial $g_j(x)$ is not received or its degree is greater than t, then F^A_{mat} resets $g_j(x) = 0$.
2. F^A_{mat} sends shares $\{g_j(\alpha_i)\}_{j\notin I;i\in I}$ to the (ideal) adversary.
3. F^A_{mat} receives the corrupted parties' polynomials $\{g_i(x)\}_{i\in I}$ from the (ideal) adversary; if a polynomial $g_i(x)$ is not received or its degree is greater than t, then F^A_{mat} resets $g_i(x) = 0$.
4. F^A_{mat} computes $\vec{Y}(x) = (Y_1(x), \ldots, Y_m(x)) = (g_1(x), \ldots, g_n(x)) \cdot A$.
5. (a) For every $j \notin I$, functionality F^A_{mat} sends party P_j the entire length-m vector $\vec{y} = \vec{Y}(0)$, together with P_j's shares $(g_1(\alpha_j), \ldots, g_n(\alpha_j))$ on the input polynomials.
   (b) In addition, functionality F^A_{mat} sends the (ideal) adversary its output: the vector of polynomials $\vec{Y}(x)$, and the corrupted parties' outputs ($\vec{y}$ together with $(g_1(\alpha_i), \ldots, g_n(\alpha_i))$, for every $i \in I$).

We have already described the protocol intended to securely compute Functionality 6.4 and motivated its security. We therefore proceed directly to the formal description of the protocol (see Protocol 6.5) and its proof of security. We recall that since all our analysis is performed in the corruption-aware model, we describe the functionality in the corruption-aware hybrid model. Thus, although the F_{VSS} functionality (Functionality 5.5) is a standard functionality, we refer to it as a "fictitiously corruption-aware" functionality, as described in Sect. 6.2.

The figure below illustrates Step 5 of Protocol 6.5. Each party receives a vector from every other party. These vectors (placed as rows) all form a matrix, whose columns are at most distance t from codewords who define the output.

Theorem 6.6. *Let $t < n/3$. Then, Protocol 6.5 is t-secure for the F^A_{mat} functionality in the F_{VSS}-hybrid model, in the presence of a static malicious adversary.*

Proof. We begin by describing the simulator $\mathcal{S}$. The simulator $\mathcal{S}$ interacts externally with the trusted party computing F^A_{mat}, and internally invokes the (hybrid model) adversary $\mathcal{A}$, hence simulating an execution of Protocol 6.5 for $\mathcal{A}$. As such, $\mathcal{S}$ has external communication with the trusted party computing F^A_{mat}, and internal communication with the real adversary $\mathcal{A}$. As part of the internal communication with $\mathcal{A}$, the simulator hands $\mathcal{A}$ messages that $\mathcal{A}$ expects to see from the honest parties in the protocol execution. In addition, $\mathcal{S}$ simulates the interaction of $\mathcal{A}$ with the ideal functionality F_{VSS} and hands it the messages it expects to receives from F_{VSS} in Protocol 6.5. $\mathcal{S}$ works as follows:

1. *$\mathcal{S}$ internally invokes $\mathcal{A}$ with the auxiliary input z.*
2. External interaction with Functionality 6.4 (Step 2): *After the honest parties send their inputs to the trusted party computing F^A_{mat}, the simulator $\mathcal{S}$ receives shares $\{g_j(\alpha_i)\}_{j\notin I,i\in I}$ on its (external) incoming communication tape from F^A_{mat}.*
3. Internal simulation of Steps 1 and 2 in Protocol 6.5: *$\mathcal{S}$ internally simulates the ideal F^n_{VSS} invocation, as follows*:

PROTOCOL 6.5 (Securely computing F^A_{mat} in the F_{VSS}-hybrid model).

- **Inputs**: Each party P_i holds a polynomial $g_i(x)$.
- **Common input**: A field description $\mathbb{F}$, n distinct nonzero elements $\alpha_1, \dots, \alpha_n \in \mathbb{F}$, and a matrix $A \in \mathbb{F}^{n\times m}$.
- **Aiding ideal functionality initialization**: Upon invocation, the trusted party computing the corruption-aware parallel VSS functionality F^n_{VSS} (i.e., Functionality 6.2) is given the set of corrupted parties I.
- **The protocol**:
  1. Each party P_i checks that its input polynomial is of degree-t; if not, it resets $g_i(x) = 0$. It then sends its polynomial $g_i(x)$ to F^n_{VSS} as its private input.
  2. Each party P_i receives the values $g_1(\alpha_i), \dots, g_n(\alpha_i)$ as output from F^n_{VSS}. If any value equals $\perp$, then P_i replaces it with 0.
  3. Denote $\vec{x}^i = (g_1(\alpha_i), \dots, g_n(\alpha_i))$. Then, each party P_i locally computes $\vec{y}^i = \vec{x}^i \cdot A$ (equivalently, for every $k = 1, \dots, m$, each P_i computes $Y_k(\alpha_i) = \sum_{\ell=1}^n g_\ell(\alpha_i) \cdot a_{\ell,k}$ where $(a_{1,k}, \dots, a_{n,k})^T$ is the kth column of A, and stores $\vec{y}^i = (Y_1(\alpha_i), \dots, Y_m(\alpha_i))$).
  4. Each party P_i sends $\vec{y}^i$ to every P_j $(1 \le j \le n)$.
  5. For every $j = 1, \dots, n$, denote the vector received by P_i from P_j by $\hat{\vec{Y}}(\alpha_j) = (\hat{Y}_1(\alpha_j), \dots, \hat{Y}_m(\alpha_j))$. (If any value is missing, it replaces it with 0. We stress that different parties may hold different vectors if a party is corrupted.) Each P_i works as follows:
     - For every $k = 1, \dots, m$, party P_i locally runs the Reed–Solomon decoding procedure (with $d = 2t+1$) on the possibly corrupted codeword $(\hat{Y}_k(\alpha_1), \dots, \hat{Y}_k(\alpha_n))$ to get the codeword $(Y_k(\alpha_1), \dots, Y_k(\alpha_n))$; see Fig. 1. It then reconstructs the polynomial $Y_k(x)$ and computes $y_k = Y_k(0)$.
- **Output**: P_i outputs $(y_1, \dots, y_m)$ as well as the shares $g_1(\alpha_i), \dots, g_n(\alpha_i)$.

$$\begin{bmatrix} \leftarrow & \hat{\vec{Y}}(\alpha_1) & \rightarrow \\ \leftarrow & \hat{\vec{Y}}(\alpha_2) & \rightarrow \\ & \vdots & \\ \leftarrow & \hat{\vec{Y}}(\alpha_n) & \rightarrow \end{bmatrix} = \begin{bmatrix} \hat{Y}_1(\alpha_1) & \cdots & \boldsymbol{\hat{Y}_k(\alpha_1)} & \cdots & \hat{Y}_m(\alpha_1) \\ \hat{Y}_1(\alpha_2) & \cdots & \boldsymbol{\hat{Y}_k(\alpha_2)} & \cdots & \hat{Y}_m(\alpha_2) \\ \vdots & & \vdots & & \vdots \\ \hat{Y}_1(\alpha_n) & \cdots & \boldsymbol{\hat{Y}_k(\alpha_n)} & \cdots & \hat{Y}_m(\alpha_n) \end{bmatrix}$$

Fig. 1. The vectors received by P_i form a matrix; error correction is run on the *columns* .

(a) *$\mathcal{S}$ simulates Step 2 of F^n_{VSS} and hands the adversary $\mathcal{A}$ the shares $\{g_j(\alpha_i)\}_{j\notin I; i\in I}$ it expects to receive (where the $g_j(\alpha_i)$ values are those received from F^A_{mat} above).*

(b) *$\mathcal{S}$ simulates Step 3 of F^n_{VSS} and receives from $\mathcal{A}$ the polynomials $\{g_i(x)\}_{i\in I}$ that $\mathcal{A}$ sends as the corrupted parties' inputs to F^n_{VSS}. If $\deg(g_i(x)) > t$, then $\mathcal{S}$ replaces it with the constant polynomial $g_i(x) = 0$.*

(c) *$\mathcal{S}$ simulates Step 4 of F^n_{VSS} and internally hands $\mathcal{A}$ the outputs $\{(g_1(\alpha_i), \dots, g_n(\alpha_i))\}_{i\in I}$; if any polynomial $g_k(x)$ is such that $\deg(g_k(x)) > t$, then $\perp$ is written instead of $g_k(\alpha_i)$.*

4. External interaction with Functionality 6.4 (Step 3): *$\mathcal{S}$ externally sends the trusted party computing F^A_{mat} the polynomials $\{g_i(x)\}_{i\in I}$ as the inputs of the corrupted parties.*

5. External interaction with Functionality 6.4 (Step 5): *At this point, the functionality* F^A_{mat} *has all the parties' inputs, and so it computes the vector of polynomials* $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot A$, *and* $\mathcal{S}$ *receives back the following output from* F^A_{mat}:
   (a) *The vector of polynomials* $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot A$,
   (b) *The output vector* $\vec{y} = (y_1, \ldots, y_m)$, *and*
   (c) *The shares* $(g_1(\alpha_i), \ldots, g_n(\alpha_i))$ *for every* $i \in I$.
6. Internal simulation of Step 4 in Protocol 6.5: *For every* $j \notin I$ *and* $i \in I$, *simulator* $\mathcal{S}$ *internally hands the adversary* $\mathcal{A}$ *the vector* $\vec{y}^j = (Y_1(\alpha_j), \ldots, Y_m(\alpha_j))$ *as the vector that honest party* P_j *sends to all other parties in Step 4 of Protocol* 6.5.
7. $\mathcal{S}$ *outputs whatever* $\mathcal{A}$ *outputs and halts.*

We now prove that for every $I \subset [n]$ with $|I| \leq t$:

$$\left\{\text{IDEAL}_{F^A_{mat},\mathcal{S}(z),I}(\vec{x})\right\}_{z\in\{0,1\}^*;\vec{x}\in\mathbb{F}^n} \equiv \left\{\text{HYBRID}^{F_{VSS}}_{\pi,\mathcal{A}(z),I}(\vec{x})\right\}_{z\in\{0,1\}^*;\vec{x}\in\mathbb{F}^n}. \tag{6.1}$$

In order to see why this holds, observe first that in the F_{VSS}-hybrid model, the honest parties actions in the protocol are *deterministic* (the randomness in the real protocol is "hidden" inside the protocol for securely computing F_{VSS}), as is the simulator $\mathcal{S}$ and the ideal functionality F^A_{mat}. Thus, it suffices to separately show that the view of the adversary is identical in both cases, and the outputs of the honest parties are identical in both cases.

By inspection of the protocol and simulation, it follows that the shares $\{(g_1(\alpha_i), \ldots, g_n(\alpha_i))\}_{i\in I}$ of the corrupted parties on the honest parties inputs and the vector of polynomials $\vec{Y}(x)$ as received by $\mathcal{S}$, provide it all the information necessary to generate the *exact* messages that the corrupted parties would receive in a real execution of Protocol 6.5. Thus, the view of the adversary is identical in the ideal execution and in the protocol execution.

Next, we show that the honest party's outputs are identical in both distributions. In order to see this, it suffices to show that the vector of polynomials $\vec{Y}(x) = (Y_1(x), \ldots, Y_m(x))$ computed by F^A_{mat} in Step 4 of the functionality specification is identical to the vector of polynomials $(Y_1(x), \ldots, Y_m(x))$ computed by each party in Step 5 of Protocol 6.5 (since this defines the outputs). First, the polynomials of the honest parties are clearly the same in both cases. Furthermore, since the adversary's view is the same it holds that the polynomials $g_i(x)$ sent by $\mathcal{S}$ to the trusted party computing F^A_{mat} are exactly the same as the polynomials used by $\mathcal{A}$ in Step 1 of Protocol 6.5. This follows from the fact that the F_{VSS} functionality is used in this step and so the polynomials of the corrupted parties obtained by $\mathcal{S}$ from $\mathcal{A}$ are exactly the same as used in the protocol. Now, observe that each polynomial $Y_k(x)$ computed by the honest parties is obtained by applying Reed–Solomon decoding to the word $(\hat{Y}_k(\alpha_1), \ldots, \hat{Y}_k(\alpha_n))$. The crucial point is that the honest parties compute the values $\hat{Y}_k(\alpha_i)$ correctly, and so for every $j \notin I$ it holds that $\hat{Y}_k(\alpha_j) = Y_k(\alpha_j)$. Thus, at least $n - t$ elements of the word $(\hat{Y}_k(\alpha_1), \ldots, \hat{Y}_k(\alpha_n))$ are "correct" and so the polynomial $Y_k(x)$ reconstructed by all the honest parties in the error correction is the same $Y_k(x)$ as computed by F^A_{mat} (irrespective of what the corrupted parties send). This completes the proof. □

6.4. *The $F_{VSS}^{subshare}$ Functionality for Sharing Shares*

Defining the Functionality We begin by defining the $F_{VSS}^{subshare}$ functionality. Informally speaking, this functionality is a way for a set of parties to verifiably give out shares of values that are themselves shares. Specifically, assume that the parties $P_1, \ldots, P_n$ hold values $f(\alpha_1), \ldots, f(\alpha_n)$, respectively, where f is a degree-t polynomial either chosen by one of the parties or generated jointly in the computation. The aim is for each party to *share its share* $f(\alpha_i)$—and not any other value—with all other parties (see Fig. 2). In the semi-honest setting, this can be achieved simply by having each party P_i choose a random polynomial $g_i(x)$ with constant term $f(\alpha_i)$ and then send each P_j the share $g_i(\alpha_j)$. However, in the malicious setting, it is necessary to force the corrupted parties to share the *correct* value and nothing else; this is the main challenge. We stress that since there are more than t honest parties, their shares fully determine $f(x)$, and so the "correct" share of a corrupted party is well defined. Specifically, letting $f(x)$ be the polynomial defined by the honest parties' shares, the aim here is to ensure that a corrupted P_i provides shares using a degree-t polynomial with constant term $f(\alpha_i)$.

The functionality definition is such that if a corrupted party P_i does not provide a valid input (i.e., it does not input a degree-t polynomial $g_i(x)$ such that $g_i(0) = f(\alpha_i)$), then $F_{VSS}^{subshare}$ defines a *new polynomial* $g_i'(x)$ that is the constant polynomial $g_i'(x) = f(\alpha_i)$ for all x, and uses $g_i'(x)$ in place of $g_i(x)$ in the outputs. This ensures that the constant term of the polynomial is always $f(\alpha_i)$, as required.

We define $F_{VSS}^{subshare}$ as a corruption-aware functionality (see Sect. 6.2). Among other reasons, this is due to the fact that the parties distributes subshares of their shares. As we described in Sect. 6.2, this enables the adversary to choose the corrupted parties' polynomials $g_i(x)$ (for $i \in I$) after seeing the corrupted parties' shares of the honest parties' polynomials (i.e., $g_j(\alpha_i)$ for every $j \notin I$ and $i \in I$).

In addition, in the protocol the parties invoke the F_{mat}^{A} functionality (Functionality 6.4) with (the transpose of) the parity-check matrix H of the appropriate Reed–Solomon code (this matrix is specified below where we explain its usage in the protocol). This adds complexity to the definition of $F_{VSS}^{subshare}$ because additional information revealed by F_{mat}^{A} to the adversary needs to be revealed by $F_{VSS}^{subshare}$ as well. In the sequel, we denote the matrix multiplication functionality with (the transpose of) the parity-check matrix H by F_{mat}^{H}. Recall that the adversary's output from F_{mat}^{H} includes $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$; see Step 5 in Functionality 6.4. Thus, in order to simulate the

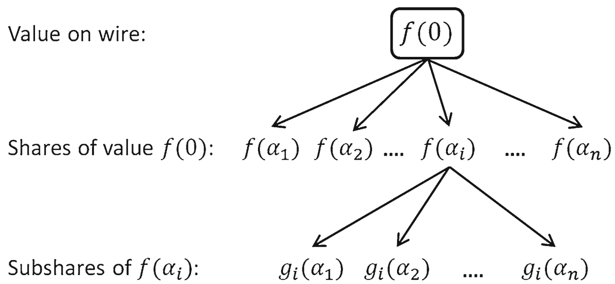

Fig. 2. The subsharing process: P_i distributes shares of its share $f(\alpha_i)$.

FUNCTIONALITY 6.7 (Functionality $F_{VSS}^{subshare}$ for subsharing shares).

$F_{VSS}^{subshare}$ receives a set of indices $I \subseteq [n]$ and works as follows:

1. $F_{VSS}^{subshare}$ receives the inputs of the honest parties $\{\beta_j\}_{j \notin I}$. Let $f(x)$ be the unique degree-t polynomial determined by the points $\{(\alpha_j, \beta_j)\}_{j \notin I}$.[9]
2. For every $j \notin I$, functionality $F_{VSS}^{subshare}$ chooses a random degree-t polynomial $g_j(x)$ under the constraint that $g_j(0) = \beta_j = f(\alpha_j)$.
3. $F_{VSS}^{subshare}$ sends the shares $\{g_j(\alpha_i)\}_{j \notin I; i \in I}$ to the (ideal) adversary.
4. $F_{VSS}^{subshare}$ receives polynomials $\{g_i(x)\}_{i \in I}$ from the (ideal) adversary; if a polynomial $g_i(x)$ is not received or if $g_i(x)$ is of degree higher than t, then $F_{VSS}^{subshare}$ sets $g_i(x) = 0$.
5. $F_{VSS}^{subshare}$ *determines the output polynomials* $g'_1(x), \ldots, g'_n(x)$*:*
   (a) For every $j \notin I$, functionality $F_{VSS}^{subshare}$ sets $g'_j(x) = g_j(x)$.
   (b) For every $i \in I$, if $g_i(0) = f(\alpha_i)$ then $F_{VSS}^{subshare}$ sets $g'_i(x) = g_i(x)$. Otherwise it sets $g'_i(x) = f(\alpha_i)$ (i.e., $g'_i(x)$ is the constant polynomial equalling $f(\alpha_i)$ everywhere).
6. (a) For every $j \notin I$, functionality $F_{VSS}^{subshare}$ sends the polynomial $g'_j(x)$ and the shares $(g'_1(\alpha_j), \ldots, g'_n(\alpha_j))$ to party P_j.
   (b) Functionality $F_{VSS}^{subshare}$ sends the (ideal) adversary the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$, where H is the parity-check matrix of the appropriate Reed-Solomon code (see below). In addition, it sends the corrupted parties' outputs $g'_i(x)$ and $(g'_1(\alpha_i), \ldots, g'_n(\alpha_i))$ for every $i \in I$.

call to F_{mat}^H, the ideal adversary needs this information. We deal with this in the same way as in Sect. 6.3 (for F_{mat}^H), by having the functionality $F_{VSS}^{subshare}$ provide the ideal adversary with the additional vector of polynomials $(g_1(x), \ldots, g_n(x)) \cdot H^T$. As we will see later, this does not interfere with our use of $F_{VSS}^{subshare}$ in order to achieve secure multiplication (which is our ultimate goal). Although it is too early to really see why this is the case, we nevertheless remark that when H is the parity-check matrix of the Reed–Solomon code, the vector $(g_1(0), \ldots, g_n(0)) \cdot H^T$ can be determined based on the corrupted parties' inputs (because we know that the honest parties' values are always "correct"), and the vector $(g_1(x), \ldots, g_n(x)) \cdot H^T$ is random under this constraint. Thus, these outputs can be simulated.

Background to Implementing $F_{VSS}^{subshare}$ Let $G \in \mathbb{F}^{(t+1)\times n}$ be the generator matrix for a (generalized) Reed–Solomon code of length $n = 3t + 1$, dimension $k = t + 1$ and distance $d = 2t+1$. In matrix notation, the encoding of a vector $\vec{a} = (a_0, \ldots, a_t) \in \mathbb{F}^{t+1}$ is given by $\vec{a} \cdot G$, where:

[9] If all of the points sent by the honest parties lie on a single degree-t polynomial, then this guarantees that $f(x)$ is the unique degree-t polynomial for which $f(\alpha_j) = \beta_j$ for all $j \notin I$. If not all the points lie on a single degree-t polynomial, then no security guarantees are obtained. However, since the honest parties all send their prescribed input, in our applications, $f(x)$ will always be as desired. This can be formalized using the notion of a partial functionality [20, Sec. 7.2]. Alternatively, it can be formalized by as follows: In the case that the condition does not hold, the ideal functionality gives all of the honest parties' inputs to the adversary and lets the adversary single-handedly determine all of the outputs of the honest parties. This makes any protocol vacuously secure (since anything can be simulated).

$$G \stackrel{\text{def}}{=} \begin{pmatrix} 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \vdots & \vdots & & \vdots \\ \alpha_1^t & \alpha_2^t & \dots & \alpha_n^t \end{pmatrix}. \tag{6.2}$$

Letting $f(x) = \sum_{\ell=0}^{t} a_\ell \cdot x^\ell$ be a degree-t polynomial, the Reed–Solomon encoding of $\vec{a} = (a_0, \dots, a_t)$ is the vector $\langle f(\alpha_1), \dots, f(\alpha_n)\rangle$. Let $H \in \mathbb{F}^{2t \times n}$ be the parity-check matrix of G; that is, H is a rank $2t$ matrix such that $G \cdot H^T = 0^{(t+1)\times 2t}$. We stress that H is full determined by $\alpha_1, \dots, \alpha_n$ and thus is a constant matrix, known to all parties. The syndrome of a word $\vec{\beta} \in \mathbb{F}^n$ is given by $S(\vec{\beta}) = \vec{\beta} \cdot H^T \in \mathbb{F}^{2t}$. A basic fact from error-correcting codes is that, for any codeword $\vec{\beta} = \vec{a} \cdot G$, it holds that $S(\vec{\beta}) = 0^{2t}$. Moreover, for every error vector $\vec{e} \in \{0, 1\}^n$, it holds that $S(\vec{\beta} + \vec{e}) = S(\vec{e})$. If $\vec{e}$ is of distance at most t from $\vec{0}$ (i.e., $\sum e_i \le t$), then it is possible to correct the vector $\vec{\beta} + \vec{e}$ and to obtain the original vector $\vec{\beta}$. An important fact is that a subprocedure of the Reed–Solomon decoding algorithm can extract the error vector $\vec{e}$ from the syndrome vector $S(\vec{e})$ alone. That is, given a possibly corrupted codeword $\vec{\gamma} = \vec{\beta} + \vec{e}$, the syndrome vector is computed as $S(\vec{\gamma}) = \vec{\gamma} \cdot H^T = S(\vec{e})$ and is given to this subprocedure, which returns $\vec{e}$. From $\vec{e}$ and $\vec{\gamma}$, the codeword $\vec{\beta}$ can be extracted easily.

The Protocol In the protocol, each party P_i chooses a random polynomial $g_i(x)$ whose constant term equals its input share β_i; let $\vec{\beta} = (\beta_1, \dots, \beta_n)$. Recall that the input shares are the shares of some polynomial $f(x)$. Thus, for all honest parties P_j it is guaranteed that $g_j(0) = \beta_j = f(\alpha_j)$. In contrast, there is no guarantee regarding the values $g_i(0)$ for corrupted P_i. Let $\vec{\gamma} = (g_1(0), \dots, g_n(0))$. It follows that $\vec{\gamma}$ is a word that is at most distance t from the vector $\vec{\beta} = (f(\alpha_1), \dots, f(\alpha_n))$, which is a Reed–Solomon codeword of length $n = 3t + 1$. Thus, it is possible to correct the word $\vec{\gamma}$ using Reed–Solomon error correction. The parties send the chosen polynomials $(g_1(x), \dots, g_n(x))$ to F_{mat}^H (i.e., Functionality 6.4 for matrix multiplication with the transpose of the parity-check matrix H described above), which hands each party P_i the output $(g_1(\alpha_i), \dots, g_n(\alpha_i))$ and $(s_1, \dots, s_{2t}) = \vec{\gamma} \cdot H^T$, where the latter equals the syndrome $S(\vec{\gamma})$ of the input vector $\vec{\gamma}$. Each party uses the syndrome in order to locally carry out error correction and obtain the error vector $\vec{e} = (e_1, \dots, e_n) = \vec{\gamma} - \vec{\beta}$. Note that $\vec{e}$ has the property that for every i it holds that $g_i(0) - e_i = f(\alpha_i)$, and $\vec{e}$ can be computed from the syndrome alone, using the subprocedure mentioned above. This error vector now provides the honest parties with all the information that they need to compute the output. Specifically, if $e_i = 0$, then this implies that P_i used a "correct" polynomial $g_i(x)$ for which $g_i(0) = f(\alpha_i)$, and so the parties can just output the shares $g_i(\alpha_j)$ that they received as output from F_{mat}^H. In contrast, if $e_i \neq 0$ then the parties know that P_i is corrupted, and can all send each other the shares $g_i(\alpha_j)$ that they received from F_{mat}^H. This enables them to reconstruct the polynomial $g_i(x)$, again using Reed–Solomon error correction, and compute $g_i(0) - e_i = f(\alpha_i)$. Thus, they obtain the actual share of the corrupted party and can set $g'_i(x) = f(\alpha_i)$, as required in the functionality definition. See Protocol 6.8 for the full description.

One issue that must be dealt with in the proof of security is due to the fact that the syndrome $\vec{\gamma} \cdot H^T$ is revealed in the protocol, and is seemingly not part of the

PROTOCOL 6.8 (Securely computing $F_{VSS}^{subshare}$ in the F_{mat}^{H}-hybrid model).

- **Inputs**: Each party P_i holds a value β_i; we assume that the points (α_j, β_j) of the honest parties all lie on a single degree-t polynomial (see the definition of $F_{VSS}^{subshare}$ above and Footnote 9 therein).
- **Common input**: A field description $\mathbb{F}$ and n distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, which determine the matrix $H \in \mathbb{F}^{2t \times n}$ which is the parity-check matrix of the Reed–Solomon code (with parameters as described above).
- **Aiding ideal functionality initialization**: Upon invocation, the trusted party computing the corruption-aware functionality F_{mat}^{H} receives the set of corrupted parties I.
- **The protocol**:
  1. Each party P_i chooses a random degree-t polynomial $g_i(x)$ under the constraint that $g_i(0) = \beta_i$
  2. The parties invoke the F_{mat}^{H} functionality (i.e., Functionality 6.4 for matrix multiplication with the transpose of the parity-check matrix H). Each party P_i inputs the polynomial $g_i(x)$ from the previous step, and receives from F_{mat}^{H} as output the shares $g_1(\alpha_i), \ldots, g_n(\alpha_i)$ and the length $2t$ vector $\vec{s} = (s_1, \ldots, s_{2t}) = (g_1(0), \ldots, g_n(0)) \cdot H^T$. Recall that $\vec{s}$ is the syndrome vector of the possible corrupted codeword $\vec{\gamma} = (g_1(0), \ldots, g_n(0))$.[10]
  3. Each party locally runs the Reed–Solomon decoding procedure using $\vec{s}$ only, and receives back an error vector $\vec{e} = (e_1, \ldots, e_n)$.
  4. For every k such that $e_k = 0$: each party P_i sets $g'_k(\alpha_i) = g_k(\alpha_i)$.
  5. For every k such that $e_k \neq 0$:
     (a) Each party P_i sends $g_k(\alpha_i)$ to every P_j.
     (b) Each party P_i receives $g_k(\alpha_1), \ldots, g_k(\alpha_n)$; if any value is missing, it sets it to 0. P_i runs the Reed–Solomon decoding procedure on the values to reconstruct $g_k(x)$.
     (c) Each party P_i computes $g_k(0)$, and sets $g'_k(\alpha_i) = g_k(0) - e_k$ (which equals $f(\alpha_k)$).
- **Output**: P_i outputs $g_i(x)$ and $g'_1(\alpha_i), \ldots, g'_n(\alpha_i)$.

output. However, recall that the adversary receives the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$ from $F_{VSS}^{subshare}$ and the syndrome is just $\vec{Y}(0)$. This is therefore easily simulated.

Theorem 6.9. *Let $t < n/3$. Then, Protocol 6.8 is t-secure for the $F_{VSS}^{subshare}$ functionality in the F_{mat}^{H}-hybrid model, in the presence of a static malicious adversary.*

Proof. We begin by describing the simulator $\mathcal{S}$. The simulator interacts externally with the ideal functionality $F_{VSS}^{subshare}$, while internally simulating the interaction of $\mathcal{A}$ with the honest parties and F_{mat}^{H}.

1. *$\mathcal{S}$ internally invokes $\mathcal{A}$ with the auxiliary input z.*
2. External interaction with Functionality 6.7 (Step 3): *After the honest parties send their polynomials $\{g_j(x)\}_{j \notin I}$ to the trusted party computing $F_{VSS}^{subshare}$, simulator $\mathcal{S}$ receives the shares $\{g_j(\alpha_i)\}_{j \notin I, i \in I}$ from $F_{VSS}^{subshare}$.*

[10] The corrupted parties also receive the vector of polynomials $(g_1(x), \ldots, g_n(x)) \cdot H^T$ as output from F_{mat}^{H}. However, in the protocol, we only specify the honest parties' instructions.

3. Internal simulation of Step 2 in Protocol 6.8: *$\mathcal{S}$ begins to internally simulate the invocation of F_{mat}^{H}.*
   (a) Internal simulation of Step 2 in Functionality 6.4: *$\mathcal{S}$ sends $\mathcal{A}$ the shares $\{g_j(\alpha_i)\}_{j\notin I, i\in I}$ as its first output from the simulated call to F_{mat}^{H} in the protocol.*
   (b) Internal simulation of Step 3 in Functionality 6.4: *$\mathcal{S}$ internally receives from $\mathcal{A}$ the polynomials $\{g_i(x)\}_{i\in I}$ that $\mathcal{A}$ sends to F_{mat}^{H} in the protocol* ().
4. External interaction with Functionality 6.7 (Step 4): *$\mathcal{S}$ externally sends the $F_{VSS}^{subshare}$ functionality the polynomials $\{g_i(x)\}_{i\in I}$ that were received in the previous step. For the rest of the execution, if* $\deg(g_i) > t$ *for some $i \in I$, $\mathcal{S}$ resets $g_i(x) = 0$.*
5. External interaction with Functionality 6.7 (Step 6b): *$\mathcal{S}$ externally receives its output from $F_{VSS}^{subshare}$, which is comprised of the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$, and the corrupted parties' outputs: polynomials $\{g_i'(x)\}_{i\in I}$ and the shares $\{g_1'(\alpha_i), \ldots, g_n'(\alpha_i)\}_{i\in I}$. Recall that $g_j'(x) = g_j(x)$ for every $j \notin I$. Moreover, for every $i \in I$, if $g_i(0) = f(\alpha_i)$ then $g_i'(x) = g_i(x)$, and $g_i'(x) = f(\alpha_j)$ otherwise.*
6. Continue internal simulation of Step 2 in Protocol 6.8 (internally simulate Step 5 of Functionality 6.4): *$\mathcal{S}$ concludes the internal simulation of F_{mat}^{H} by preparing the output that the internal $\mathcal{A}$ expects to receive from F_{mat}^{H} in the protocol, as follows*:
   (a) *$\mathcal{A}$ expects to receive the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$ from F_{mat}^{H}; however, $\mathcal{S}$ received this exact vector of polynomials from $F_{VSS}^{subshare}$ and so just hands it internally to $\mathcal{A}$.*
   (b) *In addition, $\mathcal{A}$ expects to receive the corrupted parties' outputs $\vec{y} = \vec{Y}(0)$ and the shares $\{(g_1(\alpha_i), \ldots, g_n(\alpha_i))\}_{i\in I}$. Simulator $\mathcal{S}$ can easily compute $\vec{y} = \vec{Y}(0)$ since it has the actual polynomials $\vec{Y}(x)$. In addition, $\mathcal{S}$ already received the shares $\{g_j(\alpha_i)\}_{j\notin I; i\in I}$ from $F_{VSS}^{subshare}$ and can compute the missing shares using the polynomials $\{g_i(x)\}_{i\in I}$. Thus, $\mathcal{S}$ internally hands $\mathcal{A}$ the values $\vec{y} = \vec{Y}(0)$ and $\{(g_1(\alpha_i), \ldots, g_n(\alpha_i))\}_{i\in I}$, as expected by $\mathcal{A}$.*
7. Internal simulation of Step 5a in Protocol 6.8: *$\mathcal{S}$ proceeds with the simulation of the protocol as follows. $\mathcal{S}$ computes the error vector $\vec{e} = (e_1, \ldots, e_n)$ by running the Reed–Solomon decoding procedure on the syndrome vector $\vec{s}$, that it computes as $\vec{s} = \vec{Y}(0)$ (using $\vec{Y}(x)$ that it received from $F_{VSS}^{subshare}$). Then, for every $i \in I$ for which $e_i \neq 0$ and for every $j \notin I$, $\mathcal{S}$ internally simulates P_j sending $g_i(\alpha_j)$ to all parties.*
8. *$\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.*

We now prove that for every $I \subset [n]$ with $|I| \leq t$:

$$\left\{\text{IDEAL}_{F_{VSS}^{subshare}, \mathcal{S}(z), I}(\vec{x})\right\}_{z\in\{0,1\}^*; \vec{x}\in\mathbb{F}^n} \equiv \left\{\text{HYBRID}_{\pi,\mathcal{A}(z),I}^{F_{mat}^{H}}(\vec{x})\right\}_{z\in\{0,1\}^*; \vec{x}\in\mathbb{F}^n}.$$

The main point to notice is that the simulator has enough information to perfectly emulate the honest parties' instructions. The only difference is that in a real protocol execution, the honest parties P_j choose the polynomials $g_j(x)$, whereas in an ideal execution the functionality $F_{VSS}^{subshare}$ chooses the polynomials $g_j(x)$ for every $j \notin I$. However, in both cases they are chosen at random under the constraint that $g_j(0) = \beta_j$. Thus, the distributions are identical. Apart from that, $\mathcal{S}$ has enough information to generate the exact messages that the honest parties would send. Finally, since all honest parties receive the same output from F_{mat}^{A} in the protocol execution, and this fully determines $\vec{e}$, we have that all honest parties obtain the exact same view in the protocol execution and thus all output the exact same value. Furthermore, by the error correction procedure, for every k such that $e_k \neq 0$, they reconstruct the same $g_k(x)$ sent by $\mathcal{A}$ to F_{mat}^{A} and so all define $g'_k(\alpha_j) = g_k(0) - e_k$.

A Fictitious Simulator $\mathcal{S}'$ In order to prove that the output distribution generated by $\mathcal{S}$ is identical to the output distribution of a real execution, we construct a fictitious simulator $\mathcal{S}'$ who generates the entire output distribution of both the honest parties and adversary as follows. For every $j \notin I$, simulator $\mathcal{S}'$ receives for input a *random* polynomial $g_j(x)$ under the constraint that $g_j(0) = \beta_j$. Then, $\mathcal{S}'$ invokes the adversary $\mathcal{A}$ and emulates the honest parties and the aiding functionality F_{mat}^{H} in a protocol execution with $\mathcal{A}$, using the polynomials $g_j(x)$. Finally, $\mathcal{S}'$ outputs whatever $\mathcal{A}$ outputs, together with the output of each honest party. (Note that $\mathcal{S}'$ does not interact with a trusted party and is a stand-alone machine.)

The Output Distributions It is clear that the output distribution generated by $\mathcal{S}'$ is *identical* to the output distribution of the adversary and honest parties in a real execution, since the polynomials $g_j(x)$ are chosen randomly exactly like in a real execution and the rest of the protocol is emulated by $\mathcal{S}'$ exactly according to the honest parties' instructions.

It remains to show that the output distribution generated by $\mathcal{S}'$ is *identical* to the output distribution of an ideal execution with $\mathcal{S}$ and a trusted party computing $F_{VSS}^{subshare}$. First, observe that both $\mathcal{S}'$ and $\mathcal{S}$ are *deterministic* machines. Thus, it suffices to separately show that the adversary's view is identical in both cases (given the polynomials $\{g_j(x)\}_{j\notin I}$), and the outputs of the honest parties are identical in both case (again, given the polynomials $\{g_j(x)\}_{j\notin I}$). Now, the messages generated by $\mathcal{S}$ and $\mathcal{S}'$ for $\mathcal{A}$ are identical throughout. This holds because the shares $\{g_j(\alpha_i)\}_{j\notin I;i\in I}$ of the honest parties that $\mathcal{A}$ receives from F_{mat}^{H} are the same ($\mathcal{S}$ receives them from $F_{VSS}^{subshare}$ and $\mathcal{S}'$ generates them itself from the input), as is the vector $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$ and the rest of the output from F_{mat}^{H} for $\mathcal{A}$. Finally, in Step 7 of the specification of $\mathcal{S}$ above, the remainder of the simulation after F_{mat}^{H} is carried out by running the honest parties' instructions. Thus, the messages are clearly identical and $\mathcal{A}$'s view is identical in both executions by $\mathcal{S}$ and $\mathcal{S}'$.

We now show that the output of the honest parties' as generated by $\mathcal{S}'$ is identical to their output in the ideal execution with $\mathcal{S}$ and the trusted party, given the polynomials $\{g_j(x)\}_{j\notin I}$. In the ideal execution with $\mathcal{S}$, the output of each honest party P_j is determined by the trusted party computing $F_{VSS}^{subshare}$ to be $g'_j(x)$ and $(g'_1(\alpha_j), \ldots, g'_n(\alpha_j))$. For every $j \notin I$, $F_{VSS}^{subshare}$ sets $g'_j(x) = g_j(x)$. Likewise, since the inputs of all the honest parties lie on the same degree-t polynomial, denoted f (and so $f(\alpha_j) = \beta_j$ for every $j \notin I$), we have that the error correction procedure of Reed–Solomon decoding returns an error vector $\vec{e} = (e_1, \ldots, e_n)$ such that for every k for which $g_k(0) = f(\alpha_k)$ it holds that $e_k = 0$. In particular, this holds for every $j \notin I$. Furthermore, F_{mat}^{H} guarantees that all honest parties receive the same vector $\vec{s}$ and so the error correction yields the same error vector $\vec{e}$ for every honest party. Thus, for every $j, \ell \notin I$ we have that each honest party P_ℓ sets $g'_j(\alpha_\ell) = g_j(\alpha_\ell)$, as required.

Regarding the corrupted parties' polynomials $g_i(x)$ for $i \in I$, the trusted party computing $F_{VSS}^{subshare}$ sets $g'_i(x) = g_i(x)$ if $g_i(0) = f(\alpha_i)$, and sets $g'_i(x)$ to be a constant polynomial equaling $f(\alpha_i)$ everywhere otherwise. This exact output is obtained by the honest parties for the same reasons as above: all honest parties receive the same $\vec{s}$ and thus the same $\vec{e}$. If $e_i = 0$ then all honest parties P_j set $g'_i(\alpha_j) = g_i(\alpha_j)$, whereas if $e_i \neq 0$ then the error correction enables them to reconstruct the polynomial $g_i(x)$ exactly and compute $f(\alpha_i) = g_i(0)$. Then, by the protocol every honest P_j sets its share $g'_i(\alpha_j) = f(\alpha_i) - e_i$, exactly like the trusted party. This completes the proof. □

6.5. *The F_{eval} Functionality for Evaluating a Shared Polynomial*

In the protocol for verifying the multiplication of shares presented in Sect. 6.6 (The F_{VSS}^{mult} functionality), the parties need to process "complaints" (which are claims by some of the parties that others supplied incorrect values). These complaints are processed by evaluating some shared polynomials at the point of the complaining party. Specifically, given shares $f(\alpha_1), \dots, f(\alpha_n)$, of a polynomial f, the parties need to compute $f(\alpha_k)$ for a predetermined k, without revealing anything else. (To be more exact, the shares of the honest parties define a unique degree-t polynomial f, and the parties should obtain $f(\alpha_k)$ as output.)

We begin by formally defining this functionality. The functionality is parameterized by an index k that determines at which point the polynomial is to be evaluated. In addition, we define the functionality to be corruption-aware in the sense that the polynomial is reconstructed from the honest party's inputs alone (and the corrupted parties' shares are ignored). We mention that it is possible to define the functionality so that it runs the Reed–Solomon error correction procedure on the input shares. However, defining it as we do makes it more clear that the corrupted parties can have no influence whatsoever on the output. See Functionality 6.10 for a full specification.

> **FUNCTIONALITY 6.10 (Functionality F_{eval}^k for evaluating a polynomial on α_k).**
> F_{eval}^k receives a set of indices $I \subseteq [n]$ and works as follows:
>
> 1. The F_{eval}^k functionality receives the inputs of the honest parties $\{\beta_j\}_{j \notin I}$. Let $f(x)$ be the unique degree-t polynomial determined by the points $\{(\alpha_j, \beta_j)\}_{j \notin I}$. (If not all the points lie on a single degree-t polynomial, then no security guarantees are obtained; see Footnote 9.)
> 2. (a) For every $j \notin I$, F_{eval}^k sends the output pair $(f(\alpha_j), f(\alpha_k))$ to party P_j.
>   (b) For every $i \in I$, F_{eval}^k sends the output pair $(f(\alpha_i), f(\alpha_k))$ to the (ideal) adversary, as the output of P_i.

Equivalently, in function notation, we have:

$$F_{eval}^k\Big(\beta_1, \dots, \beta_n\Big) = \Big(((f(\alpha_1), f(\alpha_k)), \dots, (f(\alpha_n), f(\alpha_k))\Big)$$

where f is the result of Reed–Solomon decoding on $(\beta_1, \dots, \beta_n)$. We remark that although each party P_i already holds $f(\alpha_i)$ as part of its input, we need the output to include this value in order to simulate (specifically, the simulator needs all of the corrupted parties' shares $\{f(\alpha_i)\}_{i \in I}$). This will not make a difference in its use, since $f(\alpha_i)$ is anyway supposed to be known to P_i.

Background We show that the share $f(\alpha_k)$ can be obtained by a linear combination of all the input shares $(\beta_1, \ldots, \beta_n)$. The parties' inputs are a vector $\vec{\beta} \stackrel{\text{def}}{=} (\beta_1, \ldots, \beta_n)$ where for every $j \notin I$ it holds that $\beta_j = f(\alpha_j)$. Thus, the parties' inputs are computed by

$$\vec{\beta} = V_{\vec{\alpha}} \cdot \vec{f}^T,$$

where $V_{\vec{\alpha}}$ is the Vandermonde matrix (see Eq. 3.2), and $\vec{f}$ is the vector of coefficients for the polynomial $f(x)$. We remark that $\vec{f}$ is of length n and is padded with zeroes beyond the $(t+1)$th entry. Let $\vec{\alpha}_k = (1, \alpha_k, (\alpha_k)^2, \ldots, (\alpha_k)^{n-1})$ be the kth row of $V_{\vec{\alpha}}$. Then the output of the functionality is

$$f(\alpha_k) = \vec{\alpha}_k \cdot \vec{f}^T.$$

We have:

$$\vec{\alpha}_k \cdot \vec{f}^T = \vec{\alpha}_k \cdot \left(V_{\vec{\alpha}}^{-1} \cdot V_{\vec{\alpha}}\right) \cdot \vec{f}^T = \left(\vec{\alpha}_k \cdot V_{\vec{\alpha}}^{-1}\right) \cdot \left(V_{\vec{\alpha}} \cdot \vec{f}^T\right) = \left(\vec{\alpha}_k \cdot V_{\vec{\alpha}}^{-1}\right) \cdot \vec{\beta}^T \tag{6.3}$$

and so there exists a vector of *fixed constants* $(\vec{\alpha}_k \cdot V_{\vec{\alpha}}^{-1})$ such that the inner product of this vector and the inputs yields the desired result. In other words, F_{eval}^k is simply a linear function of the parties' inputs.

The Protocol Since F_{eval}^k is a linear function of the parties' inputs (which are themselves shares), it would seem that it is possible to use the same methodology for securely computing F_{mat}^A (or even directly use F_{mat}^A). However, this would allow corrupted parties to input any value they wish in the computation. In contrast, the linear function that computes F_{eval}^k (i.e., the linear combination of Eq. 6.3) must be computed on the *correct* shares, where "correct" means that they *all* lie on the same degree-t polynomial. This problem is solved by having the parties subshare their input shares using a more robust input-sharing stage that guarantees that all the parties input their "correct share." Fortunately, we already have a functionality that fulfills this exact purpose: the $F_{VSS}^{subshare}$ functionality of Sect. 6.4. Therefore, the protocol consists of a *robust* input-sharing phase (i.e., an invocation of $F_{VSS}^{subshare}$), a computation phase (which is noninteractive), and an output reconstruction phase. See Protocol 6.11 for the full description.

Informally speaking, the security of the protocol follows from the fact that the parties only see subshares that reveal nothing about the original shares. Then, they see n shares of a random polynomial $Q(x)$ whose secret is the value being evaluated, enabling them to reconstruct that secret. Since the secret is obtained by the simulator/adversary as the legitimate output in the ideal model, this can be simulated perfectly.

PROTOCOL 6.11 (Securely computing F^k_{eval} in the $F^{subshare}_{VSS}$-hybrid model).

- **Inputs**: Each party P_i holds a value β_i; we assume that the points (α_j, β_j) for every honest P_j all lie on a single degree-t polynomial f (see the definition of F^k_{eval} above and Footnote 9).
- **Common input**: The description of a field $\mathbb{F}$ and n distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$.
- **Aiding ideal functionality initialization**: Upon invocation, the trusted party computing the corruption-aware functionality $F^{subshare}_{VSS}$ receives the set of corrupted parties I.
- **The protocol**:
  1. The parties invoke the $F^{subshare}_{VSS}$ functionality with each party P_i using β_i as its private input. At the end of this stage, each party P_i holds $g'_1(\alpha_i), \ldots, g'_n(\alpha_i)$, where all the $g'_i(x)$ are of degree t, and for every i it holds that $g'_i(0) = f(\alpha_i)$.
  2. Each party P_i locally computes: $Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot g'_\ell(\alpha_i)$, where $(\lambda_1, \ldots, \lambda_n) = \vec{\alpha}_k \cdot V_{\vec{\alpha}}^{-1}$. Each party P_i sends $Q(\alpha_i)$ to all P_j.
  3. Each party P_i receives all the shares $\hat{Q}(\alpha_j)$ from each other party $1 \leq j \leq n$ (if any value is missing, replace it with 0). Note that some of the parties may hold different values if a party is corrupted. Then, given the possibly corrupted codeword $(\hat{Q}(\alpha_1), \ldots, \hat{Q}(\alpha_n))$, each party runs the Reed–Solomon decoding procedure and receives the codeword $(Q(\alpha_1), \ldots, Q(\alpha_n))$. It then reconstructs $Q(x)$ and computes $Q(0)$.
- **Output**: Each party P_i outputs $(\beta_i, Q(0))$.

The main subtlety that needs to be dealt with in the proof of security is due to the fact that the $F^{subshare}_{VSS}$ functionality actually "leaks" some additional information to the adversary, beyond the vectors $(g'_1(\alpha_i), \ldots, g'_n(\alpha_i))$ for all $i \in I$. Namely, the adversary also receives the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$, where H is the parity-check matrix for the Reed–Solomon code, and $g_i(x)$ is the polynomial sent by the adversary to $F^{subshare}_{VSS}$ for the corrupted P_i and may differ from $g'_i(x)$ if the constant term of $g_i(x)$ is incorrect (for honest parties $g'_j(x) = g_j(x)$ always). The intuition as to why this vector of polynomials $\vec{Y}(x)$ can be simulated is due to the fact that the syndrome depends only on the error vector which describes the difference between the $g_i(0)$'s and $f(\alpha_i)$'s. Details follow. Let $\vec{\gamma} = (\gamma_1, \ldots, \gamma_n)$ be the inputs of the parties (where for $i \notin I$ it may be the case that $\gamma_i \neq f(\alpha_i)$). (We denote the "correct" input vector by $\vec{\beta}$—meaning $\vec{\beta} = (f(\alpha_1), \ldots, f(\alpha_n))$—and the actual inputs used by the parties by $\vec{\gamma}$.) The vector $\vec{\gamma}$ defines a word that is of distance at most t from the valid codeword $(f(\alpha_1), \ldots, f(\alpha_n))$. Thus, there exists an *error vector* $\vec{e}$ of weight at most t such that $\vec{\gamma} - \vec{e} = (f(\alpha_1), \ldots, f(\alpha_n)) = \vec{\beta}$. The syndrome function $S(\vec{x}) = \vec{x} \cdot H^T$ has the property that $S(\vec{\gamma}) = S(\vec{\beta} + \vec{e}) = S(\vec{e})$; stated differently, $(\beta_1, \ldots, \beta_n) \cdot H^T = \vec{e} \cdot H^T$. Now, $\vec{e}$ is actually fully known to the simulator. This is because for every $i \in I$ it receives $f(\alpha_i)$ from F^k_{eval}, and so when $\mathcal{A}$ sends $g_i(x)$ to $F^{subshare}_{VSS}$ in the protocol simulation, the simulator can simply compute $e_i = g_i(0) - f(\alpha_i)$. Furthermore, for all $j \notin I$, it is always the case that $e_j = 0$. Thus, the simulator can compute $\vec{e} \cdot H^T = \vec{\beta} \cdot H^T = (g_1(0), \ldots, g_n(0)) \cdot H^T = \vec{Y}(0)$ from the corrupted parties' input and output only (and the adversary's messages).

We have shown that the simulator can compute $\vec{Y}(0)$. In addition, the simulator has the values $g_1(\alpha_i), \ldots, g_n(\alpha_i)$ for every $i \in I$ and so can compute $\vec{Y}(\alpha_i) = (g_1(\alpha_i), \ldots, g_n(\alpha_i)) \cdot H^T$. As we will show, the vector of polynomials $\vec{Y}(x)$ is a series of random degree-t polynomials under the constraints $\vec{Y}(0)$ and $\{\vec{Y}(\alpha_i)\}_{i \in I}$ that $\mathcal{S}$ can compute. (Actually, when $|I| = t$ there are $t + 1$ constraints and so this vector is *fully determined*. In this case, its actually values are known to the simulator; otherwise, the simulator can just choose random polynomials that fulfill the constraints.) Finally, the same is true regarding the polynomial $Q(x)$: the simulator knows $|I| + 1$ constraints (namely $Q(0) = f(\alpha_k)$ and $Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot g'_\ell(\alpha_i)$), and can choose Q to be random under these constraints in order to simulate the honest parties sending $Q(\alpha_j)$ for every $j \notin I$. We now formally prove this.

Theorem 6.12. *Let $t < n/3$. Then, Protocol 6.11 is t-secure for the F^k_{eval} functionality in the $F^{subshare}_{VSS}$-hybrid model, in the presence of a static malicious adversary.*

Proof. The simulator interacts externally with a trusted party computing F^k_{eval}, while internally simulating the interaction of $\mathcal{A}$ with the trusted party computing $F^{subshare}_{VSS}$ and the honest parties. We have already provided the intuition behind how the simulator works, and thus proceed directly to its specification.

The Simulator $\mathcal{S}$

1. External interaction with Functionality 6.10 (Step 2b): *$\mathcal{S}$ receives the ideal adversary's output $\{(f(\alpha_i), f(\alpha_k))\}_{i \in I}$ from F^k_{eval} (recall that the corrupted parties have no input in F^k_{eval} and so it just receives output).*
2. *$\mathcal{S}$ internally invokes $\mathcal{A}$ with the auxiliary input z, and begins to simulate the protocol execution.*
3. Internal simulation of Step 1 in Protocol 6.11: *$\mathcal{S}$ internally simulates the $F^{subshare}_{VSS}$ invocations*:
   (a) Internal simulation of Step 3 in the $F^{subshare}_{VSS}$ functionality: *$\mathcal{S}$ simulates $\mathcal{A}$ receiving the shares $\{g_j(\alpha_i)\}_{j \notin I; i \in I}$: For every $j \notin I$, $\mathcal{S}$ chooses uniformly at random a polynomial $g_j(x)$ from $\mathcal{P}^{0,t}$, and sends $\mathcal{A}$ the values $\left\{g_j(\alpha_i)\right\}_{j \notin I; i \in I}$.*
   (b) Internal simulation of Step 4 in the $F^{subshare}_{VSS}$ functionality: *$\mathcal{S}$ internally receives from $\mathcal{A}$ the inputs $\{g_i(x)\}_{i \in I}$ of the corrupted parties to $F^{subshare}_{VSS}$. If for any $i \in I$, $\mathcal{A}$ did not send some polynomial $g_i(x)$, then $\mathcal{S}$ sets $g_i(x) = 0$.*
   (c) *For every $i \in I$, $\mathcal{S}$ checks that* $\deg(g_i) \leq t$ *and that $g_i(0) = f(\alpha_i)$. If this check passes, $\mathcal{S}$* sets *$g'_i(x) = g_i(x)$. Otherwise, $\mathcal{S}$ sets $g'_i(x) = f(\alpha_i)$. (Recall that $\mathcal{S}$ has $f(\alpha_i)$ from its output from F^k_{eval}.)*
   (d) *For every $j \notin I$, $\mathcal{S}$ sets $g'_j(x) = g_j(x)$.*
   (e) Internal simulation of Step 6b in the $F^{subshare}_{VSS}$ functionality: *$\mathcal{S}$ internally gives the adversary $\mathcal{A}$ the outputs, as follows*:
   i. *The vector of polynomials $\vec{Y}(x)$, which is chosen as follows*:
      - *$\mathcal{S}$ sets $(e_1, \ldots, e_n)$ such that $e_j = 0$ for every $j \notin I$, and $e_i = g_i(0) - f(\alpha_i)$ for every $i \in I$.*
      - *$\mathcal{S}$ chooses $\vec{Y}(x)$ to be a random vector of degree-t polynomials under the constraints that $\vec{Y}(0) = (e_1, \ldots, e_n) \cdot H^T$, and for every $i \in I$ it holds that $\vec{Y}(\alpha_i) = (g_1(\alpha_i), \ldots, g_n(\alpha_i)) \cdot H^T$.*

   Observe that if $|I| = t$, then all of the polynomials in $\vec{Y}(x)$ are fully determined by the above constraints.

ii. *The polynomials and values* $g'_i(x)$ *and* $\{g'_1(\alpha_i), \ldots, g'_n(\alpha_i)\}$ *for every* $i \in I$

4. $\mathcal{S}$ *simulates the sending of the shares* $Q(\alpha_j)$:
   (a) Internal simulation of Step 2 in Protocol 6.11: $\mathcal{S}$ *chooses a random polynomial* $Q(x)$ *of degree* t *under the constraints that*:
      - $Q(0) = f(\alpha_k)$.
      - *For every* $i \in I$, $Q(\alpha_i) = \sum_{\ell=1}^{n} \gamma_\ell \cdot g'_\ell(\alpha_i)$.

   (b) *For every* $j \notin I$, $\mathcal{S}$ *internally simulates honest party* P_j *sending the value* $Q(\alpha_j)$.
5. $\mathcal{S}$ *outputs whatever* $\mathcal{A}$ *outputs and halts*.

We now prove that for every $I \subseteq [n]$, such that $|I| \leq t$,

$$\left\{\text{IDEAL}_{F_{eval}^k, \mathcal{S}(z), I}(\vec{\beta})\right\}_{\vec{\beta} \in \mathbb{F}^n, z \in \{0,1\}^*} \equiv \left\{\text{HYBRID}_{\pi, \mathcal{A}(z), I}^{F_{VSS}^{subshare}}(\vec{\beta})\right\}_{\vec{\beta} \in \mathbb{F}^n, z \in \{0,1\}^*}.$$

There are three differences between the simulation with $\mathcal{S}$ and $\mathcal{A}$, and an execution of Protocol 6.11 with $\mathcal{A}$. First, $\mathcal{S}$ chooses the polynomials $g_j(x)$ to have constant terms of 0 instead of constant terms $f(\alpha_j)$ for every $j \notin I$. Second, $\mathcal{S}$ computes the vector of polynomials $\vec{Y}(x)$ based on the given constraints, rather that it being computed by $F_{VSS}^{subshare}$ based on the polynomials $(g_1(x), \ldots, g_n(x))$. Third, $\mathcal{S}$ chooses a random polynomial $Q(x)$ under the described constraints in Step 4a of $\mathcal{S}$, rather than it being computed as a function of all the polynomials $g'_1(x), \ldots, g'_n(x)$.

We eliminate these differences one at a time, by introducing three fictitious simulators.

The Fictitious Simulator $\mathcal{S}_1$ Simulator $\mathcal{S}_1$ is exactly the same as $\mathcal{S}$, except that it receives for input the values $\beta_j = f(\alpha_j)$, for every $j = 1, \ldots, n$ (rather than just $j \in I$). In addition, for every $j \notin I$, instead of choosing $g_j(x) \in_R \mathcal{P}^{0,t}$, the fictitious simulator $\mathcal{S}_1$ chooses $g_j(x) \in_R \mathcal{P}^{f(\alpha_j),t}$. We stress that $\mathcal{S}_1$ runs in the ideal model with the same trusted party running F_{eval}^k as $\mathcal{S}$, and the honest parties receive output as specified by F_{eval}^k when running with the ideal adversary $\mathcal{S}$ or $\mathcal{S}_1$.

We claim that for every $I \subseteq [n]$, such that $|I| \leq t$,

$$\left\{\text{IDEAL}_{F_{eval}^k, \mathcal{S}_1(z,\vec{\beta}), I}(\vec{\beta})\right\}_{\vec{\beta} \in \mathbb{F}^n, z \in \{0,1\}^*} \equiv \left\{\text{IDEAL}_{F_{eval}^k, \mathcal{S}(z), I}(\vec{\beta})\right\}_{\vec{\beta} \in \mathbb{F}^n, z \in \{0,1\}^*}$$

In order to see that the above holds, observe that both $\mathcal{S}$ and $\mathcal{S}_1$ can work when given the points of the inputs shares $\{g_j(\alpha_i)\}_{i \in I, j \notin I}$ and they don't actually need the polynomials themselves. Furthermore, the only difference between $\mathcal{S}$ and $\mathcal{S}_1$ is whether these points are derived from polynomials with zero constant terms, or with the "correct" ones. That is, there exists a machine $\mathcal{M}$ that receives points $\{g_j(\alpha_i)\}_{i \in I; j \notin I}$ and runs the simulation strategy with $\mathcal{A}$ while interacting with F_{eval}^k in an ideal execution, such that:

- If $g_j(0) = 0$ then the joint output of $\mathcal{M}$ and the honest parties in the ideal execution is exactly that of $\text{IDEAL}_{F_{eval}^k, \mathcal{S}(z), I}(\vec{\beta})$; i.e., an ideal execution with the original simulator.

- If $g_j(0) = f(\alpha_j)$ then the joint output of $\mathcal{M}$ and the honest parties in the ideal execution is exactly that of $\text{IDEAL}_{F^k_{eval},\mathcal{S}_1(z,\vec{\beta}),I}(\vec{\beta})$; i.e., an ideal execution with the fictitious simulator.

By Claim 3.3, the points $\{g_j(\alpha_i)\}_{i\in I; j\notin I}$ when $g_j(0) = 0$ are identically distributed to the points $\{g_j(\alpha_i)\}_{i\in I; j\notin I}$ when $g_j(0) = f(\alpha_j)$. Thus, the joint outputs of the adversary and honest parties in both simulations are identical.

The Fictitious Simulator $\mathcal{S}_2$ Simulator $\mathcal{S}_2$ is exactly the same as $\mathcal{S}_1$, except that it computes the vector of polynomials $\vec{Y}(x)$ in the same way that $F_{VSS}^{subshare}$ computes it in the real execution. Specifically, for every $j \notin I$, $\mathcal{S}_2$ chooses random polynomials $g_j(x)$ under the constraint that $g_j(0) = f(\alpha_j)$ just like honest parties. In addition, for every $i \in I$, it uses the polynomials $g_i(x)$ sent by $\mathcal{A}$. We claim that for every $I \subseteq [n]$, such that $|I| \le t$,

$$\left\{\text{IDEAL}_{F^k_{eval},\mathcal{S}_2(z,\vec{\beta}),I}(\vec{\beta})\right\}_{\vec{\beta}\in\mathbb{F}^n, z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{F^k_{eval},\mathcal{S}_1(z,\vec{\beta}),I}(\vec{\beta})\right\}_{\vec{\beta}\in\mathbb{F}^n, z\in\{0,1\}^*}$$

This follows from the aforementioned property of the syndrome function $S(\vec{x}) = \vec{x}\cdot H^T$. Specifically, let $\vec{\gamma}$ be the parties' actually inputs (for $j \notin I$ we are given that $\gamma_j = f(\alpha_j)$, but nothing is guaranteed about the value of γ_i for $i \in I$), and let $\vec{e} = (e_1, \ldots, e_n)$ be the error vector (for which $\gamma_i = f(\alpha_i) + e_i$). Then, $S(\vec{\gamma}) = S(\vec{e})$. If $|I| = t$, then the constraints fully define the vector of polynomials $\vec{Y}(x)$, and by the property of the syndrome these constraints are identical in both simulations by $\mathcal{S}_1$ and $\mathcal{S}_2$. Otherwise, if $|I| < t$, then $\mathcal{S}_1$ chooses $\vec{Y}(x)$ at random under $t+1$ constraints, whereas $\mathcal{S}_2$ computes $\vec{Y}(x)$ from the actual values. Consider each polynomial $Y_\ell(x)$ separately (for $\ell = 1, \ldots, 2t-1$). Then, for each polynomial there is a set of $t+1$ constraints and each is chosen at random under those constraints. Consider the random processes $X(s)$ and $Y(s)$ before Claim 4.4 in Sect. 4.2 (where the value "s" here for $Y_\ell(x)$ is the ℓth value in the vector $\vec{e}\cdot H^T$). Then, by Claim 4.4, the distributions are identical.

The Fictitious Simulator $\mathcal{S}_3$ Simulator $\mathcal{S}_3$ is the same as $\mathcal{S}_2$, except that it computes the polynomial $Q(x)$ using the polynomials $g'_1(x), \ldots, g'_n(x)$ instead of under the constraints. The fact that this is identical follows the exact same argument regarding $\vec{Y}_\ell(x)$ using Claim 4.4 in Sect. 4.2. Thus,

$$\left\{\text{IDEAL}_{F^k_{eval},\mathcal{S}_3(z,\vec{\beta}),I}(\vec{\beta})\right\}_{\vec{\beta}\in\mathbb{F}^n, z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{F^k_{eval},\mathcal{S}_2(z,\vec{\beta}),I}(\vec{\beta})\right\}_{\vec{\beta}\in\mathbb{F}^n, z\in\{0,1\}^*}$$

Observe that the view of $\mathcal{A}$ in $\text{IDEAL}_{F^k_{eval},\mathcal{S}_3(z,\vec{\beta}),I}(\vec{\beta})$ is exactly the same as in a real execution. It remains to show that the honest parties output the same in both this execution and in the $F_{VSS}^{subshare}$-hybrid execution of Protocol 6.11. Observe that $\mathcal{S}_3$ (and $\mathcal{S}_1/\mathcal{S}_2$) send no input to the trusted party in the ideal model. Thus, we just need to show that the honest parties always output $f(\alpha_k)$ in a real execution, when f is the polynomial defined by the input points $\{\beta_j\}_{j\notin I}$ of the honest parties. However, this follows immediately from the guarantees provided the $F_{VSS}^{subshare}$ functionality and by the Reed–Solomon error correction procedure. In particular, the only values received by the honest parties in a real execution are as follows:

1. Each honest P_j receives $g'_1(\alpha_j), \ldots, g'_n(\alpha_j)$, where it is guaranteed by $F_{VSS}^{subshare}$ that for every $i = 1, \ldots, n$ we have $g'_i(0) = f(\alpha_i)$. Thus, these values are *always* correct.
2. Each honest P_j receives values $(\hat{Q}(\alpha_1), \ldots, \hat{Q}(\alpha_n))$. Now, since $n - t$ of these values are sent by honest parties, it follows that this is a vector that is of distance at most t from the codeword $(Q(\alpha_1), \ldots, Q(\alpha_n))$. Thus, the Reed–Solomon correction procedure returns this codeword to every honest party, implying that the correct polynomial $Q(x)$ is reconstructed, and the honest party outputs $Q(0) = f(\alpha_k)$, as required.

This completes the proof. □

6.6. *The F_{VSS}^{mult} Functionality for Sharing a Product of Shares*

The F_{VSS}^{mult} functionality enables a set of parties who have *already* shared degree-t polynomials $A(x)$ and $B(x)$ to obtain shares of a *random* degree-t polynomial $C(x)$ under the constraint that $C(0) = A(0) \cdot B(0)$. See Sect. 6.1 for how this functionality is used in the overall multiplication protocol. We now formally describe the functionality.

FUNCTIONALITY 6.13 (Functionality F_{VSS}^{mult} for sharing a product of shares).
F_{VSS}^{mult} receives a set of indices $I \subseteq [n]$ and works as follows:

1. The F_{VSS}^{mult} functionality receives an input pair (a_j, b_j) from every honest party P_j $(j \notin I)$. (The dealer P_1 also has polynomials $A(x)$, $B(x)$ such that $A(\alpha_j) = a_j$ and $B(\alpha_j) = b_j$, for every $j \notin I$.)
2. F_{VSS}^{mult} computes the unique degree-t polynomials A and B such that $A(\alpha_j) = a_j$ and $B(\alpha_j) = b_j$ for every $j \notin I$ (if no such A or B exist of degree-t, then F_{VSS}^{mult} behaves differently as in Footnote 9).
3. If the dealer P_1 is honest $(1 \notin I)$, then:
   (a) F_{VSS}^{mult} chooses a random degree-t polynomial C under the constraint that $C(0) = A(0) \cdot B(0)$.
   (b) *Outputs for honest:* F_{VSS}^{mult} sends the dealer P_1 the polynomial $C(x)$, and for every $j \notin I$ it sends $C(\alpha_j)$ to P_j.
   (c) *Outputs for adversary:* F_{VSS}^{mult} sends the shares $(A(\alpha_i), B(\alpha_i), C(\alpha_i))$ to the (ideal) adversary, for every $i \in I$.
4. If the dealer P_1 is corrupted $(1 \in I)$, then:
   (a) F_{VSS}^{mult} sends $(A(x), B(x))$ to the (ideal) adversary.
   (b) F_{VSS}^{mult} receives a polynomial C as input from the (ideal) adversary.
   (c) If either $\deg(C) > t$ or $C(0) \neq A(0) \cdot B(0)$, then F_{VSS}^{mult} resets $C(x) = A(0) \cdot B(0)$; that is, the constant polynomial equalling $A(0) \cdot B(0)$ everywhere.
   (d) *Outputs for honest:* F_{VSS}^{mult} sends $C(\alpha_j)$ to P_j, for every $j \notin I$. (There is no more output for the adversary in this case.)

We remark that although the dealing party P_1 is supposed to already have $A(x)$, $B(x)$ as part of its input and each party P_i is also supposed to already have $A(\alpha_i)$ and $B(\alpha_i)$ as part of its input, this information is provided as output in order to enable simulation. Specifically, the simulator needs to know the corrupted parties "correct points" in order to properly simulate the protocol execution. In order to ensure that the simulator has this information (since the adversary is not guaranteed to have its correct points as input), it is provided by the functionality. In our use of F_{VSS}^{mult} in the multiplication protocol, this information is always known to the adversary anyway, and so there is nothing leaked by having it provided again by the functionality.

As we have mentioned, this functionality is used once the parties already hold shares of a and b (where a and b are the original shares of the dealer). The aim of the functionality is for them to now obtain shares of $a \cdot b$ via a degree-t polynomial C such that $C(0) = A(0) \cdot B(0) = a \cdot b$. We stress that a and b are not values on the wires, but rather are the *shares* of the dealing party of the original values on the wires.

The Protocol Idea Let $A(x)$ and $B(x)$ be polynomials such that $A(0) = a$ and $B(0) = b$; i.e., $A(x)$ and $B(x)$ are the polynomials used to share a and b. The idea behind the protocol is for the dealer to first define a sequence of t polynomials $D_1(x), \ldots, D_t(x)$, all of degree-t, such that $C(x) \stackrel{\text{def}}{=} A(x) \cdot B(x) - \sum_{\ell=1}^{t} x^{\ell} \cdot D_\ell(x)$ is a random degree-t polynomial with constant term equaling $a \cdot b$; recall that since each of $A(x)$ and $B(x)$ are of degree t, the polynomial $A(x) \cdot B(x)$ is of degree $2t$. We will show below how the dealer can choose $D_1(x), \ldots, D_t(x)$ such that all the coefficients from $t+1$ to $2t$ in $A(x) \cdot B(x)$ are canceled out, and the resulting polynomial $C(x)$ is of degree-t (and random). The dealer then shares the polynomials $D_1(x), \ldots, D_t(x)$, and each party locally computes its share of $C(x)$. An important property is that the constant term of $C(x)$ equals $A(0) \cdot B(0) = a \cdot b$ for *every* possible choice of polynomials $D_1(x), \ldots, D_t(x)$. This is due to the fact that each $D_\ell(x)$ is multiplied by x^ℓ (with $\ell \geq 1$) and so these do not affect $C(0)$. This guarantees that even if the dealer is malicious and does not choose the polynomials $D_1(x), \ldots, D_t(x)$ correctly, the polynomial $C(x)$ must have the correct constant term (but it will not necessarily be of degree t, as we explain below).

In more detail, after defining $D_1(x), \ldots, D_t(x)$, the dealer shares them all using F_{VSS}; this ensures that all polynomials are of degree-t and all parties have correct shares. Since each party already holds a valid share of $A(x)$ and $B(x)$, this implies that each party can *locally compute* its share of $C(x)$. Specifically, given $A(\alpha_j)$, $B(\alpha_j)$ and $D_1(\alpha_j), \ldots, D_t(\alpha_j)$, party P_j can simply compute $C(\alpha_j) = A(\alpha_j) \cdot B(\alpha_j) - \sum_{\ell=1}^{t} (\alpha_j)^\ell \cdot D_\ell(\alpha_j)$. The crucial properties are that **(a)** if the dealer is honest, then all the honest parties hold valid shares of a random degree-t polynomial with constant term $a \cdot b$, as required, and **(b)** if the dealer is malicious, all honest parties are guaranteed to hold valid shares of a polynomial with constant term $a \cdot b$ (but with no guarantee regarding the degree). Thus, all that remains is for the parties to verify that the shares that they hold for $C(x)$ define a degree-t polynomial.

It may be tempting to try to solve this problem by having the dealer share $C(x)$ using F_{VSS}, and then having each party check that the share that it received from this F_{VSS} equals the value $C(\alpha_j)$ that it computed from its shares $A(\alpha_j)$, $B(\alpha_j)$, $D_1(\alpha_j)$, $\ldots$, $D_t(\alpha_j)$. To be precise, denote by $C(\alpha_j)$ the share received from F_{VSS}, and denote by $C'(\alpha_j)$ the share obtained from computing $A(\alpha_j) \cdot B(\alpha_j) - \sum_{\ell=1}^{t} (\alpha_j)^\ell \cdot D_\ell(\alpha_j)$. If $C'(\alpha_j) \neq C(\alpha_j)$, then like in Protocol 5.6 for VSS, the parties broadcast complaints. If more than t complaints are broadcast then the honest parties know that the dealer is corrupted (more than t complaints are needed since the corrupted parties can falsely complain when the dealer is honest). They can then broadcast their input shares to reconstruct $A(x)$, $B(x)$ and all define their output shares to be $a \cdot b = A(0) \cdot B(0)$. Since F_{VSS} guarantees that the polynomial shared is of degree-t and we already know that the computed polynomial has the correct constant term, this seems to provide the guarantee that the parties hold shares of a degree-t polynomial with constant term $A(0) \cdot B(0)$. However, the assumption that $t+1$ correct shares (as is guaranteed by viewing at most t complaints) determines that the polynomial computed is of degree-t, or that the polynomial shared with VSS has constant term $A(0) \cdot B(0)$ is *false*. This is due to the fact that it is possible for the dealer to define the polynomials $D_1(x), \ldots, D_t(x)$ so that $C(x)$ is a degree $2t$ polynomial that agrees with some other degree-t polynomial $C'(x)$ on up to $2t$ of the honest parties' points α_j, but for which $C'(0) \neq a \cdot b$. A malicious dealer can then share $C'(x)$ using F_{VSS} and no honest parties would detect any cheating.[11] Observe that at least one honest party would detect cheating and would complain (because $C(x)$ can only agree with $C'(x)$ on $2t$ of the points, and there are at least $2t+1$ honest parties). However, this is not enough to act upon because, as described, when the dealer is honest up to t of the parties could present fake complaints because they are malicious.

We solve this problem by having the parties *unequivocally verify every complaint* to check if it is legitimate. If the complaint is legitimate, then they just reconstruct the initial shares a and b and all output the constant share $a \cdot b$. In contrast, if the complaint is not legitimate, the parties just ignore it. This guarantees that if no honest parties complain (legitimately), then the degree-t polynomial $C'(x)$ shared using F_{VSS} agrees with the computed polynomial $C(x)$ on at least $2t+1$ points. Since $C(x)$ is of degree at most $2t$, this implies that $C(x) = C'(x)$ and so it is actually of degree-t, as required.

In order to unequivocally verify complaints, we use the F_{eval}^k functionality defined in Sect. 6.5 to reconstruct all of the input shares $A(\alpha_k)$, $B(\alpha_k)$, $D_1(\alpha_k), \ldots, D_t(\alpha_k)$ and $C'(\alpha_k)$ of the complainant. Given all of the these shares, all the parties can locally compute $C'(\alpha_k) = A(\alpha_k) \cdot B(\alpha_k) - \sum_{\ell=1}^{t} (\alpha_k)^\ell \cdot D_\ell(\alpha_k)$ and check whether $C'(\alpha_k) = C(\alpha_k)$ or not. If equality holds, then the complaint is false, and is ignored. Otherwise, the complaint is valid (meaning that the dealer is corrupted), and the parties proceed to publicly reconstruct $a \cdot b$. This methodology therefore provides a way to fully verify whether a complaint was valid or not. (We remark that the parties are guaranteed to have valid shares of all the polynomials $C'(x)$, $D_1(x), \ldots, D_t(x)$ since they are shared using F_{VSS}, and also shares of $A(x)$ and $B(x)$ by the assumption on the inputs. Thus, they can use F_{eval}^k to obtain all of the values $A(\alpha_k)$, $B(\alpha_k)$, $D_1(\alpha_k), \ldots, D_t(\alpha_k)$, and $C'(\alpha_k)$, as required.)

Observe that if the dealer is honest, then no party can complain legitimately. In addition, when the dealer is honest and an illegitimate complaint is sent by a corrupted party, then this complaint is verified using F_{eval} which reveals nothing more than the complainants shares. Since the complainant in this case is corrupted, and so its share is

[11] An alternative strategy could be to run the verification strategy of Protocol 5.6 for VSS on the shares $C(\alpha_j)$ that the parties computed in order to verify that $\{C(\alpha_j)\}_{j=1}^{n}$ define a degree-t polynomial. The problem with this strategy is that if $C(x)$ is not a degree-t polynomial, then the protocol for F_{VSS} *changes* the points that the parties receive so that it is a degree-t polynomial. However, in this process, the constant term of the resulting polynomial may also change. Thus, there will no longer be any guarantee that the honest parties hold shares of a polynomial with the correct constant term.

already known to the adversary, this reveals no additional information.

Constructing the Polynomial $C(x)$ As we have mentioned above, the protocol works by having the dealer choose t polynomials $D_1(x), \ldots, D_t(x)$ that are specially designed so that $C(x) = A(x) \cdot B(x) - \sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x)$ is a *uniformly distributed* polynomial in $\mathcal{P}^{a \cdot b, t}$, where $a = A(0)$ and $b = B(0)$. We now show how the dealer chooses these polynomials. The dealer first defines the polynomial $D(x)$:

$$D(x) \stackrel{\text{def}}{=} A(x) \cdot B(x) = a \cdot b + d_1 x + \cdots + d_{2t} x^{2t}$$

($D(x)$ is of degree $2t$ since both $A(x)$ and $B(x)$ are of degree-t). Next it defines the polynomials:

$$\begin{aligned}
D_t(x) &= r_{t,0} + r_{t,1}x + \cdots + r_{t,t-1}x^{t-1} + d_{2t}x^t \\
D_{t-1}(x) &= r_{t-1,0} + r_{t-1,1}x + \cdots + r_{t-1,t-1}x^{t-1} + \left(d_{2t-1} - r_{t,t-1}\right) \cdot x^t \\
D_{t-2}(x) &= r_{t-2,0} + r_{t-2,1}x + \cdots + r_{t-2,t-1}x^{t-1} + \left(d_{2t-2} - r_{t-1,t-1} - r_{t,t-2}\right) \cdot x^t \\
&\vdots \\
D_1(x) &= r_{1,0} + r_{1,1}x + \cdots r_{1,t-1}x^{t-1} + \left(d_{t+1} - r_{t,1} - r_{t-1,2} - \ldots - r_{2,t-1}\right) x^t
\end{aligned}$$

where all $r_{i,j} \in_R \mathbb{F}$ are random values, and the d_i values are the coefficients from $D(x) = A(x) \cdot B(x)$.[12] That is, in each polynomial $D_{\ell}(x)$ all coefficients are random expect for the t^{th} coefficient, which equals the $(t+\ell)$th coefficient of $D(x)$. More exactly, for $1 \leq \ell \leq t$ polynomial $D_{\ell}(x)$ is defined by:

$$D_{\ell}(x) = r_{\ell,0} + r_{\ell,1} \cdot x + \cdots + r_{\ell,t-1} \cdot x^{t-1} + \left(d_{t+\ell} - \sum_{m=\ell+1}^{t} r_{m,t+\ell-m}\right) \cdot x^t$$

and the polynomial $C(x)$ is computed by:

$$C(x) = D(x) - \sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x).$$

Before proceeding, we show that when the polynomials $D_1(x), \ldots, D_t(x)$ are chosen in this way, it holds that $C(x)$ is a degree-t polynomial with constant term $A(0) \cdot B(0) = a \cdot b$. Specifically, the coefficients in $D(x)$ for powers greater than t cancel out. For every polynomial $D_{\ell}(x)$, we have that: $D_{\ell}(x) = r_{\ell,0} + r_{\ell,1} \cdot x + \cdots + r_{\ell,t-1} \cdot x^{t-1} + R_{\ell,t} \cdot x^t$, where

$$R_{\ell,t} = d_{t+\ell} - \sum_{m=\ell+1}^{t} r_{m,t+\ell-m}. \tag{6.4}$$

[12] The *naming convention* for the $r_{i,j}$ values is as follows. In the first $t-1$ coefficients, the first index in every $r_{i,j}$ value is the index of the polynomial and the second is the place of the coefficient. That is, $r_{i,j}$ is the jth coefficient of polynomial $D_i(x)$. The values for the t^{th} coefficient are used in the other polynomials as well, and are chosen to cancel out; see below.

Table 1. Coefficients of the polynomial $\sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x)$.

	x	x^2	x^3	$\dots$	x^t	x^{t+1}	x^{t+2}	$\dots$	x^{2t-2}	x^{2t-1}	x^{2t}
D_t					$r_{t,0}$	$r_{t,1}$	$r_{t,2}$	$\dots$	$r_{t,t-2}$	$r_{t,t-1}$	$R_{t,t}$
D_{t-1}				$\dots$	$r_{t-1,1}$	$r_{t-1,2}$	$r_{t-1,3}$	$\dots$	$r_{t-1,t-1}$	$R_{t-1,t}$	
D_{t-2}				$\dots$	$r_{t-2,2}$	$r_{t-2,3}$	$r_{t-2,4}$	$\dots$	$R_{t-2,t}$		
$\vdots$				$\iddots$	$\vdots$	$\vdots$	$\vdots$	$\iddots$			
D_3			$r_{3,0}$	$\dots$	$r_{3,t-3}$	$r_{3,t-2}$	$r_{3,t-1}$	$\dots$			
D_2		$r_{2,0}$	$r_{2,1}$	$\dots$	$r_{2,t-2}$	$r_{2,t-1}$	$R_{2,t}$				
D_1	$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$\dots$	$r_{1,t-1}$	$R_{1,t}$					

(Observe that the sum of the *indices* (i, j) of the $r_{i,j}$ values inside the sum is *always* $t + \ell$ exactly.) We now analyze the structure of the polynomial $\sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x)$. First, observe that it is a polynomial of degree $2t$ with constant term 0 (the constant term is 0 since $\ell \geq 1$). Next, the coefficient of the monomial x^{ℓ} is the *sum* of the coefficients of the ℓth column in Table 1; in the table, the coefficients of the polynomial $D_{\ell}(x)$ are written in the ℓth row and are shifted ℓ places to the right since $D_{\ell}(x)$ is multiplied by x^{ℓ}.

We will now show that for every $k = 1, \dots, t$ the coefficient of the monomial x^{t+k} in the polynomial $\sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x)$ equals d_{t+k}. Now, the sum of the $(t + k)$th column of the above table (for $1 \leq k \leq t$) is

$$R_{k,t} + r_{k+1,t-1} + r_{k+2,t-2} + \cdots + r_{t,k} = R_{k,t} + \sum_{m=k+1}^{t} r_{m,t+k-m}.$$

Combining this with the definition of $R_{k,t}$ in Eq. (6.4), we have that all of the $r_{i,j}$ values cancel out, and the sum of the $(t+k)$th column is just d_{t+k}. We conclude that the $(t+k)$th coefficient of $C(x) = D(x) - \sum_{\ell=1}^{t} x^{\ell} \cdot D_{\ell}(x)$ equals $d_{t+k} - d_{t+k} = 0$, and thus $C(x)$ is of degree t, as required. The fact that $C(0) = a \cdot b$ follows immediately from the fact that each $D_{\ell}(x)$ is multiplied by x^{ℓ} and so this does not affect the constant term of $D(x)$. Finally, observe that the coefficients of $x, x^2, \dots, x^t$ are all random (since for every $i = 1, \dots, t$ the value $r_{i,0}$ appears only in the coefficient of x^i). Thus, the polynomial $C(x)$ also has random coefficients everywhere except for the constant term.

The protocol See Protocol 6.14 for a full specification in the $(F_{VSS}, F_{eval}^1, \dots, F_{eval}^n)$-hybrid model. From here on, we write the F_{eval}-hybrid model to refer to all n functionalities $F_{eval}^1, \dots, F_{eval}^n$.

We have the following theorem:

Theorem 6.15. *Let $t < n/3$. Then, Protocol 6.14 is t-secure for the F_{VSS}^{mult} functionality in the (F_{VSS}, F_{eval})-hybrid model, in the presence of a static malicious adversary.*

PROTOCOL 6.14 (Securely computing F_{VSS}^{mult} in the F_{VSS}-F_{eval}-hybrid model).

- **Input**:
  1. The dealer P_1 holds two degree-t polynomials A and B.
  2. Each party P_i holds a pair of shares a_i and b_i such that $a_i = A(\alpha_i)$ and $b_i = B(\alpha_i)$.
- **Common input**: A field description $\mathbb{F}$ and n distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$.
- **Aiding ideal functionality initialization**: Upon invocation, the trusted party computing the (fictitiously corruption-aware) functionality F_{VSS} and the corruption-aware functionality F_{eval} receives the set of corrupted parties I.
- **The protocol**:
  1. *Dealing phase*:
     (a) The dealer P_1 defines the degree-$2t$ polynomial $D(x) = A(x) \cdot B(x)$; denote $D(x) = a \cdot b + \sum_{\ell=1}^{2t} d_\ell \cdot x^\ell$.
     (b) P_1 chooses t^2 values $\{r_{k,j}\}$ uniformly and independently at random from $\mathbb{F}$, where $k = 1, \ldots, t$, and $j = 0, \ldots, t-1$.
     (c) For every $\ell = 1, \ldots, t$, the dealer P_1 defines the polynomial $D_\ell(x)$:
     $$D_\ell(x) = \left(\sum_{m=0}^{t-1} r_{\ell,m} \cdot x^m \right) + \left(d_{\ell+t} - \sum_{m=\ell+1}^{t} r_{m,t+\ell-m} \right) \cdot x^t .$$
     (d) P_1 computes the polynomial:
     $$C(x) = D(x) - \sum_{\ell=1}^{t} x^\ell \cdot D_\ell(x).$$
     (e) P_1 invokes F_{VSS} as dealer with input $C(x)$; each party P_i receives $C(\alpha_i)$.
     (f) P_1 invokes F_{VSS} as dealer with input $D_\ell(x)$ for every $\ell = 1, \ldots, t$; each party P_i receives $D_\ell(\alpha_i)$.
  2. *Verify phase*: Each party P_i works as follows:
     (a) If any of the $C(\alpha_i)$, $D_\ell(\alpha_i)$ values equals $\perp$ then P_i proceeds to the *reject phase* (note that if one honest party received $\perp$ then all did).
     (b) Otherwise, P_i computes $c_i' = a_i \cdot b_i - \sum_{\ell=1}^{t} (\alpha_i)^\ell \cdot D_\ell(\alpha_i)$. If $c_i' \neq C(\alpha_i)$ then P_i broadcasts $(\mathsf{complaint}, i)$.
     (c) If any party P_k broadcast $(\mathsf{complaint}, k)$ then go to the *complaint resolution phase*. Otherwise, go to the output stage (and output $C(\alpha_i)$).
  3. *Complaint resolution phase*: Set $\mathsf{reject} = \mathsf{false}$. Then, run the following for every $(\mathsf{complaint}, k)$ message:
     (a) Run $t+3$ invocations of F_{eval}^k: in the first (resp., second) invocation each party P_i inputs a_i (resp., b_i), in the third invocation each P_i inputs $C(\alpha_i)$, and in the $(\ell+3)$th invocation each P_i inputs $D_\ell(\alpha_i)$ for $\ell = 1, \ldots, t$.
     (b) Let $A(\alpha_k), B(\alpha_k), \tilde{C}(\alpha_k), \tilde{D}_1(\alpha_k), \ldots, \tilde{D}_t(\alpha_k)$ be the respective outputs that all parties receive from the invocations. Compute $\tilde{C}'(\alpha_k) = A(\alpha_k) \cdot B(\alpha_k) - \sum_{\ell=1}^{t} {\alpha_k}^\ell \cdot \tilde{D}_\ell(\alpha_k)$. (We denote these polynomials by $\tilde{C}, \tilde{D}_\ell, \ldots$ since if the dealer is not honest they may differ from the specified polynomials above.)
     (c) If $\tilde{C}(\alpha_k) \neq \tilde{C}'(\alpha_k)$, then set $\mathsf{reject} = \mathsf{true}$.

     If $\mathsf{reject} = \mathsf{false}$, then go to the output stage (and output $C(\alpha_i)$). Else, go to the reject phase.
  4. *Reject phase*:
     (a) Every party P_i broadcasts the pair (a_i, b_i). Let $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$ be the broadcast values (where zero is used for any value not broadcast). Then, P_i computes $A'(x)$ and $B'(x)$ to be the outputs of Reed–Solomon decoding on $\vec{a}$ and $\vec{b}$, respectively.
     (b) Every party P_i sets $C(\alpha_i) = A'(0) \cdot B'(0)$.
- **Output**: Every party P_i outputs $C(\alpha_i)$.

Proof. We separately prove the security of the protocol when the dealer is honest and when the dealer is corrupted.

Case 1: The Dealer P_1 is Honest The simulator interacts externally with F_{VSS}^{mult}, while internally simulating the interaction of $\mathcal{A}$ with the honest parties and F_{VSS}, F_{eval} in Protocol 6.14. Since the dealer is honest, in all invocations of F_{VSS} the adversary has no inputs to these invocations and just receives shares. Moreover, as specified in the F_{VSS}^{mult} functionality, the ideal adversary/simulator $\mathcal{S}$ has no input to F_{VSS}^{mult} and it just receives the correct input shares $(A(\alpha_i), B(\alpha_i))$ and the output shares $C(\alpha_i)$ for every $i \in I$. The simulator $\mathcal{S}$ simulates the view of the adversary by choosing random degree-t polynomials $D_2(x), \ldots, D_t(x)$, and then choosing $D_1(x)$ randomly under the constraint that for every $i \in I$ it holds that

$$\alpha_i \cdot D_1(\alpha_i) = A(\alpha_i) \cdot B(\alpha_i) - C(\alpha_i) - \sum_{\ell=2}^{t} {\alpha_i}^{\ell} \cdot D_\ell(\alpha_i).$$

This computation yields $D_1(\alpha_i), \ldots, D_t(\alpha_i)$ of the correct distribution since

$$C(x) = D(x) - \sum_{\ell=1}^{t} x^{\ell} \cdot D_\ell(x) = A(x) \cdot B(x) - x \cdot D_1(x) - \sum_{\ell=2}^{t} x^{\ell} \cdot D_\ell(x)$$

implying that

$$x \cdot D_1(x) = A(x) \cdot B(x) - C(x) - \sum_{\ell=2}^{t} x^{\ell} \cdot D_\ell(x).$$

As we will see, the polynomials $D_\ell(x)$ chosen by an honest dealer have the same distribution as those chosen by $\mathcal{S}$ (they are random under the constraint that $C(\alpha_i) = A(\alpha_i) \cdot B(\alpha_i) - \sum_{\ell=1}^{t} (\alpha_i)^{\ell} \cdot D_\ell(\alpha_i)$ for all $i \in I$). In order to simulate the complaints, observe that no honest party broadcasts a complaint. Furthermore, for every (complaint, i) value broadcast by a corrupted P_i ($i \in I$), the complaint resolution phase can easily be simulated since $\mathcal{S}$ knows the correct values $\tilde{A}(\alpha_i) = A(\alpha_i)$, $\tilde{B}(\alpha_i) = B(\alpha_i)$, $\tilde{C}(\alpha_i) = C(\alpha_i)$. Furthermore, for every $\ell = 1, \ldots, t$, $\mathcal{S}$ uses $\tilde{D}_\ell(\alpha_i) = D_\ell(\alpha_i)$ as chosen initially in the simulation as the output from F_{eval}^{i}. We now formally describe the simulator.

The Simulator $\mathcal{S}$

1. *$\mathcal{S}$ internally invokes the adversary $\mathcal{A}$ with the auxiliary input z.*
2. External interaction with Functionality 6.13 (Step 3c): *$\mathcal{S}$ externally receives from F_{VSS}^{mult} the values $(A(\alpha_i), B(\alpha_i), C(\alpha_i))$ for every $i \in I$. (Recall that the adversary has no input to F_{VSS}^{mult} in the case that the dealer is honest.)*
3. *$\mathcal{S}$ chooses $t-1$ random degree-t polynomials $D_2(x), \ldots, D_t(x)$.*
4. *For every $i \in I$, $\mathcal{S}$ computes*:

$$D_1(\alpha_i) = (\alpha_i)^{-1} \cdot \left(A(\alpha_i) \cdot B(\alpha_i) - C(\alpha_i) - \sum_{\ell=2}^{t} (\alpha_i)^{\ell} \cdot D_\ell(\alpha_i) \right)$$

5. Internal simulation of Steps 1e and 1f in Protocol 6.14: *$\mathcal{S}$ simulates the F_{VSS} invocations, and simulates every corrupted party P_i (for every $i \in I$) internally receiving outputs $C(\alpha_i)$, $D_1(\alpha_i), \ldots, D_t(\alpha_i)$ from F_{VSS} in the respective invocations.*
6. Internal simulation of Steps 2 and 3 in Protocol 6.14: *For every $k \in I$ for which $\mathcal{A}$ instructs the corrupted party P_k to broadcast a* $(\mathsf{complaint}, k)$ *message, $\mathcal{S}$ simulates the* complaint resolution phase (*Step 3 of Protocol* 6.14) *by internally simulating the $t+3$ invocations of F^k_{eval}: For every $i \in I$, the simulator internally hands the adversary $(A(\alpha_i), A(\alpha_k))$, $(B(\alpha_i), B(\alpha_k))$, $(C(\alpha_i), C(\alpha_k))$ and $\{(D_\ell(\alpha_i), D_\ell(\alpha_k))\}_{\ell=1}^t$ as P_i's outputs from the respective invocation of F^k_{eval}.*
7. *$\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and halts.*

We prove that for every for every $I \subseteq [n]$, every $z \in \{0,1\}^*$ and all vectors of inputs $\vec{x}$,

$$\left\{\text{IDEAL}_{F^{mult}_{VSS},\mathcal{S}(z),I}(\vec{x})\right\} \equiv \left\{\text{HYBRID}^{F_{VSS},F_{eval}}_{\pi,\mathcal{A}(z),I}(\vec{x})\right\}.$$

We begin by showing that the outputs of the honest parties are distributed identically in an ideal execution with $\mathcal{S}$ and in a real execution of the protocol with $\mathcal{A}$ (the protocol is actually run in the (F_{VSS}, F_{eval})-hybrid model, but we say "real" execution to make for a less cumbersome description). Then, we show that the view of the adversary is distributed identically, when the output of the honest parties is given.

The Honest Parties' Outputs We analyze the distribution of the output of honest parties. Let the inputs of the honest parties be shares of the degree-t polynomials $A(x)$ and $B(x)$. Then, in the ideal model the trusted party chooses a polynomial $C(x)$ that is distributed uniformly at random in $\mathcal{P}^{A(0)\cdot B(0),t}$, and sends each party P_j the output $(A(\alpha_j), B(\alpha_j), C(\alpha_j))$.

In contrast, in a protocol execution, the honest dealer chooses $D_1(x), \ldots, D_t(x)$ and then derives $C(x)$ from $D(x) = A(x) \cdot B(x)$ and the polynomial $D_1(x), \ldots, D_t(x)$; see Steps 1a to 1d in Protocol 6.14. It is immediate that the polynomial C computed by the dealer in the protocol is such that $C(0) = A(0) \cdot B(0)$ and that each honest party P_j outputs $C(\alpha_j)$. This is due to the fact that, since the dealer is honest, all the complaints that are broadcasted are resolved with the result that $\tilde{C}(\alpha_k) \neq \tilde{C}'(\alpha_k)$, and so the *reject phase* is never reached. Thus, the honest parties output shares of a polynomial $C(x)$ with the correct constant term. It remains to show that $C(x)$ is of degree-t and is *uniformly distributed* in $\mathcal{P}^{A(0)\cdot B(0),t}$. In the discussion above, we have already shown that $\deg(C) \leq t$, and that every coefficient of $C(x)$ is random, except for the constant term.

We conclude that $C(x)$ as computed by the honest parties is uniformly distributed in $\mathcal{P}^{A(0)\cdot B(0),t}$ and so the distribution over the outputs of the honest parties in a real protocol execution is identical to their output in an ideal execution.

The Adversary'S View We now show that the view of the adversary is identical in the real protocol and ideal executions, given the honest parties' inputs and outputs. Fix the honest parties' input shares $(A(\alpha_j), B(\alpha_j))$ and output shares $C(\alpha_j)$ for every $j \notin I$.

Observe that these values fully determine the degree-t polynomials $A(x)$, $B(x)$, $C(x)$ since there are more than t points. Now, the view of the adversary in a real protocol execution is comprised of the shares

$$\{D_1(\alpha_i)\}_{i\in I}, \ldots, \{D_t(\alpha_i)\}_{i\in I}, \{C(\alpha_i)\}_{i\in I} \tag{6.5}$$

received from the F_{VSS} invocations, and of the messages from the complaint resolution phase. In the complaint resolution phase, the adversary merely sees some subset of the shares in Eq. (6.5). This is due to the fact that in this corruption case where the dealer is honest, only corrupted parties complain. Since $C(x)$ is fixed (since we are conditioning over the input and output of the honest parties), we have that it suffices for us to show that the $D_1(\alpha_i), \ldots, D_t(\alpha_i)$ values are identically distributed in an ideal execution and in a real protocol execution.

Formally, denote by $D_1^S(x), \ldots, D_t^S(x)$ the polynomials chosen by $\mathcal{S}$ in the simulation, and by $D_1(x), \ldots, D_t(x)$ the polynomials chosen by the honest dealer in a protocol execution. Then, it suffices to prove that

$$\begin{aligned}&\left\{D_1^S(\alpha_i), \ldots, D_t^S(\alpha_i) \mid A(x), B(x), C(x)\right\}_{i\in I}\\ &\quad\equiv \left\{D_1(\alpha_i), \ldots, D_t(\alpha_i) \mid A(x), B(x), C(x)\right\}_{i\in I}\end{aligned} \tag{6.6}$$

In order to prove this, we show that for every $\ell = 1, \ldots, t$,

$$\begin{aligned}&\left\{D_\ell^S(\alpha_i) \mid A(x), B(x), C(x), D_{\ell+1}^S(\alpha_i), \ldots, D_t^S(\alpha_i)\right\}_{i\in I}\\ &\quad\equiv \left\{D_\ell(\alpha_i) \mid A(x), B(x), C(x), D_{\ell+1}(\alpha_i), \ldots, D_t(\alpha_i)\right\}_{i\in I}.\end{aligned} \tag{6.7}$$

Combining all of the above (from $\ell = t$ downto $\ell = 1$), we derive Eq. (6.6).

We begin by proving Eq. (6.7) for $\ell > 1$, and leave the case of $\ell = 1$ for the end. Let $\ell \in \{2, \ldots, t\}$. It is clear that the points $\{D_\ell^S(\alpha_i)\}_{i\in I}$ are uniformly distributed, because the simulator $\mathcal{S}$ chose $D_\ell^S(x)$ uniformly at random, and independently of $A(x)$, $B(x)$, $C(x)$ and $D_{\ell+1}^S(x), \ldots, D_t^S(x)$. In contrast, in the protocol, there seems to be dependence between $D_\ell(x)$ and the polynomials $A(x)$, $B(x)$, $C(x)$ and $D_{\ell+1}(x), \ldots, D_t(x)$. In order to see that this is not a problem, note that

$$D_\ell(x) = r_{\ell,0} + r_{\ell,1} \cdot x + \cdots + r_{\ell,t-1} \cdot x^{t-1} + \left(d_{\ell+t} - \sum_{m=\ell+1}^{t} r_{m,t+\ell-m}\right) \cdot x^t$$

where the values $r_{\ell,0}, \ldots, r_{\ell,t-1}$ are all random and do *not* appear in any of the polynomials $D_{\ell+1}(x), \ldots, D_t(x)$, nor of course in $A(x)$ or $B(x)$; see Table 1. Thus, the only dependency is in the t^{th} coefficient (since the values $r_{m,t+\ell-m}$ appear in the polynomials $D_{\ell+1}(x), \ldots, D_t(x)$). However, by Claim 3.4 it holds that if $D_\ell(x)$ is a degree-t polynomial in which its *first* t coefficients are uniformly distributed, then any t points $\{D_\ell(\alpha_i)\}_{i\in I}$ are uniformly distributed. Finally, regarding the polynomial $C(x)$ observe that the m^{th} coefficient of $C(x)$, for $1 \le m \le t$ in the real protocol includes the random value $r_{1,m-1}$ (that appears in no other polynomials; see Table 1), and the constant term is always $A(0) \cdot B(0)$. Since $r_{1,m-1}$ are random and appear only in $D_1(x)$, this implies that $D_\ell(x)$ is independent of $C(x)$. This completes the proof of Eq. (6.7) for $\ell > 1$.

It remains now to prove Eq. (6.7) for the case $\ell = 1$; i.e., to show that the points $\{D_1^S(\alpha_i)\}_{i \in I}$ and $\{D_1(\alpha_i)\}_{i \in I}$ are identically distributed, conditioned on $A(x), B(x), C(x)$ and all the points $\{D_2(\alpha_i), \ldots, D_t(\alpha_i)\}_{i \in I}$. Observe that the polynomial $D_1(x)$ chosen by the dealer in the real protocol is fully determined by $C(x)$ and $D_2(x), \ldots, D_t(x)$. Indeed, an equivalent way of describing the dealer is for it to choose all $D_2(x), \ldots, D_t(x)$ as before, to choose $C(x)$ uniformly at random in $\mathcal{P}^{a \cdot b, t}$ and then to choose $D_1(x)$ as follows:

$$D_1(x) = x^{-1} \cdot \left(A(x) \cdot B(x) - C(x) - \sum_{k=2}^{t} x^k \cdot D_k(x) \right). \tag{6.8}$$

Thus, once $D_2(x), \ldots, D_t(x), A(x), B(x), C(x)$ are fixed, the polynomial $D_1(x)$ is fully determined. Likewise, in the simulation, the points $\{D_1(\alpha_i)\}_{i \in I}$ are fully determined by $\{D_2(\alpha_i), \ldots, D_t(\alpha_i), A(\alpha_i), B(\alpha_i), C(\alpha_i)\}_{i \in I}$. Thus, the actual values $\{D_1(\alpha_i)\}_{i \in I}$ are the same in the ideal execution and real protocol execution, when conditioning as in Eq. (6.7). (Intuitively, the above proof shows that the distribution over the polynomials in a real execution is identical to choosing a random polynomial $C(x) \in \mathcal{P}^{A(0) \cdot B(0), t}$ and random points $D_2(\alpha_i), \ldots, D_t(\alpha_i)$, and then choosing random polynomials $D_2(x), \ldots, D_t(x)$ that pass through these points, and determining $D_1(x)$ so that Eq. (6.8) holds.)

We conclude that the view of the corrupted parties in the protocol is identically distributed to the adversary's view in the ideal simulation, given the outputs of the honest parties. Combining this with the fact that the outputs of the honest parties are identically distributed in the protocol and ideal executions, we conclude that the joint distributions of the adversary's output and the honest parties' outputs in the ideal and real executions are identical.

Case 2—The Dealer is Corrupted In the case that the dealer P_1 is corrupted, the ideal adversary sends a polynomial $C(x)$ to the trusted party computing F_{VSS}^{mult}. If the polynomial is of degree at most t and has the constant term $A(0) \cdot B(0)$, then this polynomial determines the output of the honest parties. Otherwise, the polynomial $C(x)$ determining the output shares of the honest parties is the constant polynomial equaling $A(0) \cdot B(0)$ everywhere.

Intuitively, the protocol is secure in this corruption case because any deviation by a corrupted dealer from the prescribed instructions is unequivocally detected in the verify phase via the F_{eval} invocations. Observe also that in the (F_{VSS}, F_{eval})-hybrid model, the adversary receives no messages from the honest parties except for those sent in the complaint phase. However, the adversary already knows the results of these complaints in any case. In particular, since the adversary (in the ideal model) knows $A(x)$ and $B(x)$, and it dealt the polynomials $C(x), D_1(x), \ldots, D_t(x)$, it knows exactly where a complaint will be sent and it knows the values revealed by the F_{eval}^k calls.

We now formally describe the simulator (recall that the ideal adversary receives the polynomials $A(x)$, $B(x)$ from F_{VSS}^{mult}; this is used to enable the simulation).

The Simulator $\mathcal{S}$

1. *$\mathcal{S}$ internally invokes $\mathcal{A}$ with the auxiliary input z.*
2. External interaction with Functionality 6.13 (Step 4a): *$\mathcal{S}$ externally receives the polynomials $A(x)$, $B(x)$ from F_{VSS}^{mult}.*
3. Internal simulation of Steps 1e and 1fin Protocol 6.14: *$\mathcal{S}$ internally receives the polynomials $C(x), D_1(x), \ldots, D_t(x)$ that $\mathcal{A}$ instructs the corrupted dealer to use in the F_{VSS} invocations.*
4. *If $\deg(C) > t$ or if $\deg(D_\ell) > t$ for some $1 \leq \ell \leq t$, then $\mathcal{S}$ proceeds to Step 8 below (simulating reject).*
5. Internal simulation of Steps 2 and 3 in Protocol 6.14: *For every $k \notin I$ such that $C(\alpha_k) \neq A(\alpha_k) \cdot B(\alpha_k) - \sum_{\ell=1}^{t} (\alpha_k)^\ell \cdot D_\ell(\alpha_k)$, the simulator $\mathcal{S}$ simulates the honest party P_k broadcasting the message* ($\mathsf{complaint}$, k). *Then, $\mathcal{S}$ internally simulates the "complaint resolution phase." In this phase, $\mathcal{S}$ uses the polynomials $A(x)$, $B(x)$, $C(x)$ and $D_1(x), \ldots, D_t(x)$ in order to compute the values output in the F_{eval}^k invocations. If there exists such a $k \notin I$ as above, then $\mathcal{S}$ proceeds to Step 8 below.*
6. *For every* ($\mathsf{complaint}$, k) *message (with $k \in I$) that was internally broadcast by the adversary $\mathcal{A}$ in the name of a corrupted party P_k, the simulator $\mathcal{S}$ uses the polynomials $A(x)$, $B(x)$, $C(x)$ and $D_1(x), \ldots, D_t(x)$ in order to compute the values output in the F_{eval}^k invocations, as above. Then, if there exists an $i \in I$ such that $C(\alpha_k) \neq A(\alpha_k) \cdot B(\alpha_k) - \sum_{\ell=1}^{t} (\alpha_k)^\ell \cdot D_\ell(\alpha_k)$, simulator $\mathcal{S}$ proceeds to Step 8 below.*
7. External interaction with Functionality 6.13 (Step 4b): *If $\mathcal{S}$ reaches this point, then it externally sends the polynomial $C(x)$ obtained from $\mathcal{A}$ above to F_{VSS}^{mult}. It then skips to Step 9 below.*
8. Internal simulation of Step 4 in Protocol 6.14: *$\mathcal{S}$ simulates a reject, as follows*:
   (a) *$\mathcal{S}$ externally sends $\hat{C}(x) = x^{t+1}$ to the trusted party computing F_{VSS}^{mult} (i.e., $\mathcal{S}$ sends a polynomial $\hat{C}$ such that* $\deg(\hat{C}) > t$).
   (b) *$\mathcal{S}$ internally simulates every honest party P_j broadcasting $a_j = A(\alpha_j)$ and $b_j = B(\alpha_j)$ as in the reject phase.*
9. *$\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and halts.*

The simulator obtains $A(x)$, $B(x)$ from F_{VSS}^{mult} and can therefore compute the actual inputs $a_j = A(\alpha_j)$ and $b_j = B(\alpha_j)$ held by all honest parties P_j ($j \notin I$). Therefore, the view of the adversary in the simulation is clearly identical to its view in a real execution. We now show that the output of the honest parties in the ideal model and in a real protocol execution are identical, *given* the view of the corrupted parties/adversary. We have two cases in the ideal model/simulation:

1. *Case 1—$\mathcal{S}$ does not simulate reject ($\mathcal{S}$ does not run Step 8)* This case occurs if
   (a) All the polynomials $C(x), D_1(x), \ldots, D_t(x)$ are of degree-t, *and*

(b) For every $j \notin I$, it holds that $C(\alpha_j) = A(\alpha_j) \cdot B(\alpha_j) - \sum_{\ell=1}^{t} (\alpha_j)^\ell \cdot D_\ell(\alpha_j)$, *and*

(c) If any corrupt P_i broadcast (complaint, i) then $C(\alpha_i) = A(\alpha_i) \cdot B(\alpha_i) - \sum_{\ell=1}^{t} (\alpha_i)^\ell \cdot D_\ell(\alpha_i)$.

The polynomials obtained by $\mathcal{S}$ from $\mathcal{A}$ in the simulation are the same polynomials used by $\mathcal{A}$ in the F_{VSS} calls in the real protocol. Thus, in this case, in the protocol execution it is clear that each honest party P_j will output $C(\alpha_j)$.

In contrast, in the ideal model, each honest P_j will outputs $C(\alpha_j)$ as long as $\deg(C) \le t$ and $C(0) = A(0) \cdot B(0)$. Now, let $C'(x) = A(x) \cdot B(x) - \sum_{\ell=1}^{t} x^\ell \cdot D_\ell(x)$. By the definition of C' and the fact that each $D_\ell(x)$ is guaranteed to be of degree-t, we have that $C'(x)$ is of degree at most $2t$. Furthermore, in this case, we know that for every $j \notin I$, it holds that $C(\alpha_j) = A(\alpha_j) \cdot B(\alpha_j) - \sum_{\ell=1}^{t} (\alpha_j)^\ell \cdot D_\ell(\alpha_j) = C'(\alpha_j)$. Thus, $C(x) = C'(x)$ on at least $2t+1$ points $\{\alpha_j\}_{j \notin I}$. This implies that $C(x) = C'(x)$, and in particular $C(0) = C'(0)$. Since $C'(0) = A(0) \cdot B(0)$ *irrespective* of the choice of the polynomials $D_1(x), \ldots, D_t(x)$, we conclude that $C(0) = A(0) \cdot B(0)$. The fact that $C(x)$ is of degree-t follows from the conditions of this case. Thus, we conclude that in the ideal model, every honest party P_j also outputs $C(\alpha_j)$, exactly as in a protocol execution.

2. *Case 2—$\mathcal{S}$ simulates reject ($\mathcal{S}$ runs Step 8)* This case occurs if any of (a), (b) or (c) above do not hold. When this occurs in a protocol execution, all honest parties run the reject phase in the real execution and output the value $A(0) \cdot B(0)$. Furthermore, in the ideal model, in any of these cases the simulator $\mathcal{S}$ sends the polynomial $\hat{C}(x) = x^{t+1}$ to F_{VSS}^{mult}. Now, upon input of $C(x)$ with $\deg(C) > t$, functionality F_{VSS}^{mult} sets $C(x) = A(0) \cdot B(0)$ and so all honest parties output the value $A(0) \cdot B(0)$, exactly as in a protocol execution.

This concludes the proof. □

6.7. *The F_{mult} Functionality and Its Implementation*

We are finally ready to show how to securely compute the product of shared values, in the presence of malicious adversaries. As we described in the high-level overview in Sect. 6.1, the multiplication protocol works by first having each party share subshares of its two input shares (using $F_{VSS}^{subshare}$), and then share the product of the shares (using F_{VSS}^{mult} and the subshares obtained from $F_{VSS}^{subshare}$). Finally, given shares of the product of each party's two input shares, a sharing of the product of the *input values* is obtained via a local computation of a linear function by each party.

The Functionality We begin by defining the multiplication functionality for the case of malicious adversaries. In the semi-honest setting, the F_{mult} functionality was defined as follows:

$$F_{mult}\Big((f_a(\alpha_1), f_b(\alpha_1)), \ldots, (f_a(\alpha_n), f_b(\alpha_n))\Big) = \Big(f_{ab}(\alpha_1), \ldots, f_{ab}(\alpha_n)\Big)$$

where f_{ab} is a random degree-t polynomial with constant term $f_a(0) \cdot f_b(0) = a \cdot b$.

In the malicious setting, we need to define the functionality with more care. First, the corrupted parties are able to influence the output and determine the shares of the corrupted parties in the output polynomial. In order to see why this is the case, recall that the multiplication works by the parties running F_{VSS}^{mult} multiple times (in each invocation a different party plays the dealer) and then computing a linear function of the subshares obtained. Since each corrupted party can choose which polynomial $C(x)$ is used in F_{VSS}^{mult} when it is the dealer, the adversary can single-handedly determine the shares of the corrupted parties in the final polynomial that hides the product of the values. This is similar to the problem that arises when running F_{VSS} in parallel, as described in Sect. 6.2. In addition, there is no dealer, and the corrupted parties have no control over the resulting polynomial, beyond choosing their own shares. We model this by defining the F_{mult} multiplication functionality as a reactive corruption-aware functionality. See Functionality 6.16 for a full specification.

FUNCTIONALITY 6.16 (Functionality F_{mult} for emulating a multiplication gate).

F_{mult} receives a set of indices $I \subseteq [n]$ and works as follows:

1. The F_{mult} functionality receives the inputs of the honest parties $\{(\beta_j, \gamma_j)\}_{j \notin I}$. Let $f_a(x), f_b(x)$ be the unique degree-t polynomials determined by the points $\{(\alpha_j, \beta_j)\}_{j \notin I}$, $\{(\alpha_j, \gamma_j)\}_{j \notin I}$, respectively. (If such polynomials do not exist then no security is guaranteed; see Footnote 9.)
2. F_{mult} sends $\{(f_a(\alpha_i), f_b(\alpha_i))\}_{i \in I}$ to the (ideal) adversary.[13]
3. F_{mult} receives points $\{\delta_i\}_{i \in I}$ from the (ideal) adversary (if some δ_i is not received, then it is set to equal 0).
4. F_{mult} chooses a random degree-t polynomial $f_{ab}(x)$ under the constraints that:
   (a) $f_{ab}(0) = f_a(0) \cdot f_b(0)$, and
   (b) For every $i \in I$, $f_{ab}(\alpha_i) = \delta_i$.

   (such a degree-t polynomial always exists since $|I| \leq t$).
5. The functionality F_{mult} sends the value $f_{ab}(\alpha_j)$ to every honest party P_j $(j \notin I)$.

Before proceeding, we remark that the F_{mult} functionality is sufficient for use in circuit emulation. Specifically, the only difference between it and the definition of multiplication in the semi-honest case is the ability of the adversary to determine its own values. However, since f_{ab} is of degree-t, the ability of $\mathcal{A}$ to determine t values of f_{ab} reveals nothing about $f_{ab}(0) = a \cdot b$. A formal proof of this is given in Sect. 7.

The Protocol Idea We are now ready to show how to multiply in the $F_{VSS}^{subshare}$ and F_{VSS}^{mult} hybrid model. Intuitively, the parties first distribute subshares of their shares and subshares of the product of their shares, using $F_{VSS}^{subshare}$ and F_{VSS}^{mult}, respectively. Note that F_{VSS}^{mult} assumes that the parties already hold correct subshares,; this is achieved by first running $F_{VSS}^{subshare}$ on the input shares. Next, we use the method from Gennaro et al. [19] to have the parties directly compute shares of the *product of the values* on the input wires, from the subshares of the *product of their shares*. This method is based on the following

[13] As with F_{eval} and F_{VSS}^{mult}, the simulator needs to receive the correct shares of the corrupted parties in order to simulate, and so this is also received as output. Since this information is anyway given to the corrupted parties, this makes no difference to the use of the functionality for secure computation.

observation. Let $f_a(x)$ and $f_b(x)$ be two degree-t polynomials such that $f_a(0) = a$ and $f_b(0) = b$, and let $h(x) = f_a(x) \cdot f_b(x) = a \cdot b + h_1 \cdot x + h_2 \cdot x^2 + \cdots + h_{2t} \cdot x^{2t}$. Letting $V_{\vec{\alpha}}$ be the Vandermonde matrix for $\vec{\alpha}$, and recalling that $V_{\vec{\alpha}}$ is invertible, we have that

$$V_{\vec{\alpha}} \cdot \begin{pmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{pmatrix} \quad \text{and so} \quad \begin{pmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = V_{\vec{\alpha}}^{-1} \cdot \begin{pmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{pmatrix}.$$

Let $\lambda_1, \ldots, \lambda_n$ be the first row of $V_{\vec{\alpha}}^{-1}$. It follows that

$$\begin{aligned} a \cdot b &= \lambda_1 \cdot h(\alpha_1) + \cdots + \lambda_n \cdot h(\alpha_n) \\ &= \lambda_1 \cdot f_a(\alpha_1) \cdot f_b(\alpha_1) + \cdots + \lambda_n \cdot f_a(\alpha_n) \cdot f_b(\alpha_n). \end{aligned}$$

Thus the parties simply need to compute a linear combination of the products $f_a(\alpha_\ell) \cdot f_b(\alpha_\ell)$ for $\ell = 1, \ldots, n$. Using $F_{VSS}^{subshare}$ and F_{VSS}^{mult}, as described above, the parties first distribute random shares of the values $f_a(\alpha_\ell) \cdot f_b(\alpha_\ell)$, for every $\ell = 1, \ldots, n$. That is, let $C_1(x), \ldots, C_n(x)$ be random degree-t polynomials such that for every ℓ it holds that $C_\ell(0) = f_a(\alpha_\ell) \cdot f_b(\alpha_\ell)$; the polynomial $C_\ell(x)$ is shared using F_{VSS}^{mult} where P_ℓ is the dealer (since P_ℓ's input shares are $f_a(\alpha_\ell)$ and $f_b(\alpha_\ell)$). Then, the result of the sharing via F_{VSS}^{mult} is that each party P_i holds $C_1(\alpha_i), \ldots, C_n(\alpha_i)$. Thus, each P_i can locally compute $Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C_\ell(\alpha_i)$ and we have that the parties hold shares of the polynomial $Q(x) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C_\ell(x)$. By the fact that $C_\ell(0) = f_a(\alpha_\ell) \cdot f_b(\alpha_\ell)$ for every ℓ, it follows that

$$Q(0) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C_\ell(0) = \sum_{\ell=1}^{n} \lambda_\ell \cdot f_a(\alpha_\ell) \cdot f_b(\alpha_\ell) = a \cdot b. \tag{6.9}$$

Furthermore, since all the $C_\ell(x)$ polynomials are of degree-t, the polynomial $Q(x)$ is also of degree-t, implying that the parties hold a valid sharing of $a \cdot b$, as required. Full details of the protocol are given in Protocol 6.17.

The correctness of the protocol is based on the above discussion. Intuitively, the protocol is secure since the invocations of $F_{VSS}^{subshare}$ and F_{VSS}^{mult} provide shares to the parties that reveal nothing. However, recall that the adversary's output from $F_{VSS}^{subshare}$ includes the vector of polynomials $\vec{Y}(x) = (g_1(x), \ldots, g_n(x)) \cdot H^T$, where $g_1, \ldots, g_n$ are the polynomials defining the parties' input shares, and H is the parity-check matrix of the appropriate Reed–Solomon code; see Sect. 6.4. In the context of Protocol 6.17, this means that the adversary also obtains the vectors of polynomials $\vec{Y}_A(x) = (A_1(x), \ldots, A_n(x)) \cdot H^T$ and $\vec{Y}_B(x) = (B_1(x), \ldots, B_n(x)) \cdot H^T$. Thus, we must also show that these vectors can be generated by the simulator for the adversary. The strategy for doing so is exactly as in the simulation of F_{eval} in Sect. 6.5. We prove the following:

PROTOCOL 6.17. (Computing F_{mult} in the $(F_{VSS}^{subshare}, F_{VSS}^{mult})$-hybrid model).

- **Input**: Each party P_i holds a_i, b_i, where $a_i = f_a(\alpha_i)$, $b_i = f_b(\alpha_i)$ for some polynomials $f_a(x)$, $f_b(x)$ of degree t, which hide a, b, respectively. (If not all the points lie on a single degree-t polynomial, then no security guarantees are obtained. See Footnote 9.)
- **Common input**: A field description $\mathbb{F}$ and n distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$.
- **Aiding ideal functionality initialization**: Upon invocation, the trusted party computing the corruption-aware functionalities $F_{VSS}^{subshare}$ and F_{VSS}^{mult} receives the set of corrupted parties I.
- **The protocol**:
  1. The parties invoke the $F_{VSS}^{subshare}$ functionality with each party P_i using a_i as its private input. Each party P_i receives back shares $A_1(\alpha_i), \ldots, A_n(\alpha_i)$, and a polynomial $A_i(x)$. (Recall that for every i, the polynomial $A_i(x)$ is of degree-t and $A_i(0) = f_a(\alpha_i) = a_i$.)
  2. The parties invoke the $F_{VSS}^{subshare}$ functionality with each party P_i using b_i as its private input. Each party P_i receives back shares $B_1(\alpha_i), \ldots, B_n(\alpha_i)$, and a polynomial $B_i(x)$.
  3. For every $i = 1, \ldots, n$, the parties invoke the F_{VSS}^{mult} functionality as follows:
     (a) *Inputs*: In the ith invocation, party P_i plays the dealer. All parties P_j $(1 \le j \le n)$ send F_{VSS}^{mult} their shares $A_i(\alpha_j), B_i(\alpha_j)$.
     (b) *Outputs*: The dealer P_i receives $C_i(x)$ where $C_i(x) \in_R \mathcal{P}^{A_i(0)\cdot B_i(0),t}$, and every party P_j $(1 \le j \le n)$ receives the value $C_i(\alpha_j)$.
  4. At this stage, each party P_i holds values $C_1(\alpha_i), \ldots, C_n(\alpha_i)$, and locally computes $Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C_\ell(\alpha_i)$, where $(\lambda_1, \ldots, \lambda_n)$ is the first row of the matrix $V_{\vec{\alpha}}^{-1}$.
- **Output**: Each party P_i outputs $Q(\alpha_i)$.

Theorem 6.18. *Let $t < n/3$. Then, Protocol 6.17 is t-secure for the F_{mult} functionality in the $(F_{VSS}^{subshare}, F_{VSS}^{mult})$-hybrid model, in the presence of a static malicious adversary.*

Proof. As we have mentioned, in our analysis here we assume that the inputs of the honest parties all lie on two polynomials of degree t; otherwise (vacuous) security is immediate as described in Footnote 9. We have already discussed the motivation behind the protocol and therefore proceed directly to the simulator. The simulator externally interacts with the trusted party computing F_{mult}, internally invokes the adversary $\mathcal{A}$, and simulates the honest parties in Protocol 6.17 and the interaction with the $F_{VSS}^{subshare}$ and F_{VSS}^{mult} functionalities.

The Simulator $\mathcal{S}$

1. *$\mathcal{S}$ internally invokes $\mathcal{A}$ with the auxiliary input z.*
2. External interaction with Functionality 6.16 (Step 2): *$\mathcal{S}$ externally receives from the trusted party computing F_{mult} the values $(f_a(\alpha_i), f_b(\alpha_i))$, for every $i \in I$.*
3. Internal simulation of Step 1 in Protocol 6.17: *$\mathcal{S}$ simulates the first invocation of $F_{VSS}^{subshare}$, as follows*:
   (a) *For every $j \notin I$, $\mathcal{S}$ chooses a polynomial $A_j(x) \in_R \mathcal{P}^{0,t}$ uniformly at random.*
   (b) Internal simulation of Step 3 in Functionality 6.7: *$\mathcal{S}$ internally hands $\mathcal{A}$ the values $\{A_j(\alpha_i)\}_{j \notin I; i \in I}$ as if coming from $F_{VSS}^{subshare}$.*
   (c) Internal simulation of Step 4 in Functionality 6.7: *$\mathcal{S}$ internally receives from $\mathcal{A}$ a set of polynomials $\{A_i(x)\}_{i \in I}$ (i.e., the inputs of the corrupted parties to*

$F_{VSS}^{subshare}$). *If any polynomial is missing, then $\mathcal{S}$ sets it to be the constant polynomial* 0.

(d) Internal simulation of Step 5b in Functionality 6.7: *For every $i \in I$, $\mathcal{S}$ performs the following checks*:

i. *$\mathcal{S}$ checks that $A_i(0) = f_a(\alpha_i)$, and*
ii. *$\mathcal{S}$ checks that the degree of $A_i(x)$* is t.

If both checks pass, then it sets $A'_i(x) = A_i(x)$. Otherwise, $\mathcal{S}$ sets $A'_i(x)$ to be the constant polynomial that equals $f_a(\alpha_i)$ everywhere (*recall that $\mathcal{S}$ received $f_a(\alpha_i)$ from F_{mult} in Step* 6.7 *and so can carry out this check and set the output to be these values if necessary*).
For every $j \notin I$, $\mathcal{S}$ sets $A'_j(x) = A_j(x)$.

(e) *$\mathcal{S}$ computes the vector of polynomials $\vec{Y}_A(x)$ that $\mathcal{A}$ expects to receive from $F_{VSS}^{subshare}$ (in a real execution, $\vec{Y}_A(x) = (A_1(x), \ldots, A_n(x)) \cdot H^T$). In order to do this, $\mathcal{S}$ first computes the error vector $\vec{e}^A = (e_1^A, \ldots, e_n^A)$ as follows: for every $j \notin I$ it sets $e_j^A = 0$, and for every $i \in I$ it sets $e_i^A = A_i(0) - f(\alpha_i)$. Then, $\mathcal{S}$ chooses a vector of random polynomials $\vec{Y}_A(x) = (Y_1(x), \ldots, Y_n(x))$ under the constraints that* **(a)** *$\vec{Y}_A(0) = (e_1^A, \ldots, e_n^A) \cdot H^T$, and* **(b)** *$\vec{Y}_A(\alpha_i) = (A_1(\alpha_i), \ldots, A_n(\alpha_i)) \cdot H^T$ for every $i \in I$.*

(f) Internal simulation of Step 6b in Functionality 6.7: *$\mathcal{S}$ internally hands $\mathcal{A}$ its output from $F_{VSS}^{subshare}$. Namely, it hands the adversary $\mathcal{A}$ the polynomials $\{A'_i(x)\}_{i \in I}$, the shares $\{A'_1(\alpha_i), \ldots, A'_n(\alpha_i)\}_{i \in I}$, and the vector of polynomials $\vec{Y}_A(x)$ computed above.*

4. Internal simulation of Step 1 in Protocol 6.17 (cont.): *$\mathcal{S}$ simulates the second invocation of $F_{VSS}^{subshare}$. This simulation is carried out in an identical way using the points $\{f_b(\alpha_i)\}_{i \in I}$. Let $B_1(x), \ldots, B_n(x)$ and $B'_1(x), \ldots, B'_n(x)$ be the polynomials used by $\mathcal{S}$ in the simulation of this step* (*and so $\mathcal{A}$ receives from $\mathcal{S}$ as output from $F_{VSS}^{subshare}$ the values $\{B'_i(x)\}_{i \in I}$, $\{B'_1(\alpha_i), \ldots, B'_n(\alpha_i)\}_{i \in I}$ and $\vec{Y}_B(x)$ computed analogously to above*).
At this point $\mathcal{S}$ holds a set of degree-t polynomials $\{A'_\ell(x), B'_\ell(x)\}_{\ell \in [n]}$, where for every $j \notin I$ it holds that $A'_j(0) = B'_j(0) = 0$, and for every $i \in I$ it holds that $A'_i(0) = f_a(\alpha_i)$ and $B'_i(0) = f_b(\alpha_i)$.

5. Internal simulation of Step 3 in Protocol 6.17*: For every $j \notin I$, $\mathcal{S}$ simulates the F_{VSS}^{mult} invocation where the honest party P_j is dealer*:

(a) *$\mathcal{S}$ chooses a uniformly distributed polynomial $C'_j(x) \in_R \mathcal{P}^{0,t}$.*
(b) *$\mathcal{S}$ internally hands the adversary $\mathcal{A}$ the shares $\{(A'_j(\alpha_i), B'_j(\alpha_i), C'_j(\alpha_i))\}_{i \in I}$, as if coming from F_{VSS}^{mult}* (*Step 3c in Functionality* 6.13).

6. Internal simulation of Step 3 in Protocol 6.17 (cont.): For every $i \in I$, $\mathcal{S}$ simulates the F_{VSS}^{mult} invocation where the corrupted party P_i is dealer:

(a) Internal simulation of Step 4a of Functionality 6.13: *$\mathcal{S}$ internally hands the adversary $\mathcal{A}$ the polynomials $(A'_i(x), B'_i(x))$ as if coming from F_{VSS}^{mult}.*
(b) Internal simulation of Step 4b of Functionality 6.13: *$\mathcal{S}$ internally receives from $\mathcal{A}$ the input polynomial $C_i(x)$ of the corrupted dealer that $\mathcal{A}$ sends to F_{VSS}^{mult}.*

i. *If the input is a polynomial* C_i *such that* $\deg(C_i) \leq t$ *and* $C_i(0) = A'_i(0) \cdot B'_i(0) = f_a(\alpha_i) \cdot f_b(\alpha_i)$, *then* $\mathcal{S}$ *sets* $C'_i(x) = C_i(x)$.

ii. *Otherwise,* $\mathcal{S}$ *sets* $C'_i(x)$ *to be the constant polynomial equaling* $f_a(\alpha_i) \cdot f_b(\alpha_i)$ *everywhere.*

At this point, $\mathcal{S}$ holds polynomials $C'_1(x), \ldots, C'_n(x)$, where for every $j \notin I$ it holds that $C'_j(0) = 0$ and for every $i \in I$ it holds that $C'_j(0) = f_a(\alpha_i) \cdot f_b(\alpha_i)$.

7. External interaction with Functionality 6.16 (Step 3): *For every* $i \in I$, *the simulator* $\mathcal{S}$ *computes* $Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(\alpha_i)$, *where* $C'_1(x), \ldots, C'_n(x)$ *are as determined by* $\mathcal{S}$ *above, and sends the set* $\{Q(\alpha_i)\}_{i \in I}$ *to the* F_{mult} *functionality* (*this is the set* $\{\delta_i\}_{i \in I}$ *in Step 3 of Functionality* 6.16).
8. $\mathcal{S}$ *outputs whatever* $\mathcal{A}$ *outputs.*

The differences between the simulation with $\mathcal{S}$ and $\mathcal{A}$, and a real execution of Protocol 6.17 with $\mathcal{A}$ are as follows. First, for every $j \notin I$, $\mathcal{S}$ chooses the polynomials $A'_j(x)$, $B'_j(x)$, and $C'_j(x)$ to have constant terms of 0 instead of constant terms $f_a(\alpha_j)$, $f_b(\alpha_j)$, and $f_a(\alpha_j) \cdot f_b(\alpha_j)$, respectively. Second, the vectors $\vec{Y}_A(x)$ and $\vec{Y}_B(x)$ are computed by $\mathcal{S}$ using the error vector, and not using the actual polynomials $A_1(x), \ldots, A_n(x)$ and $B_1(x), \ldots, B_n(x)$, as computed by $F_{VSS}^{subshare}$ in the protocol execution. Third, in an ideal execution the output shares are generated by F_{mult} choosing a random degree-t polynomial $f_{ab}(x)$ under the constraints that $f_{ab}(0) = f_a(0) \cdot f_b(0)$, and $f_{ab}(\alpha_i) = \delta_i$ for every $i \in I$. In contrast, in a real execution, the output shares are derived from the polynomial $Q(x) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(x)$. Apart from these differences, the executions are identical since $\mathcal{S}$ is able to run the checks of the $F_{VSS}^{subshare}$ and F_{VSS}^{mult} functionalities exactly as they are specified.

Our proof proceeds by constructing intermediate fictitious simulators to bridge between the real and ideal executions.

The Fictitious Simulator $\mathcal{S}_1$ Let $\mathcal{S}_1$ be exactly the same as $\mathcal{S}$, except that it receives for input the values $f_a(\alpha_j)$, $f_b(\alpha_j)$, for every $j \notin I$. Then, instead of choosing $A'_j(x) \in_R \mathcal{P}^{0,t}$, $B'_j(x) \in_R \mathcal{P}^{0,t}$, and $C'_j(x) \in_R \mathcal{P}^{0,t}$, the fictitious simulator $\mathcal{S}_1$ chooses $A'_j(x) \in_R \mathcal{P}^{f_a(\alpha_j),t}$, $B'_j(x) \in_R \mathcal{P}^{f_b(\alpha_j),t}$, and $C'_j(x) \in_R \mathcal{P}^{f_a(\alpha_j) \cdot f_b(\alpha_j),t}$. We stress that $\mathcal{S}_1$ runs in the ideal model with the same trusted party running F_{mult} as $\mathcal{S}$, and the honest parties receive output as specified by F_{mult} when running with the ideal adversary $\mathcal{S}$ or $\mathcal{S}_1$.

The Ideal Executions with $\mathcal{S}$ *and* $\mathcal{S}_1$ We begin by showing that the joint output of the adversary and honest parties is identical in the original simulation by $\mathcal{S}$ and the fictitious simulation by $\mathcal{S}_1$. That is,

$$\left\{\mathrm{IDEAL}_{F_{mult},\mathcal{S}(z),I}(\vec{x})\right\}_{\vec{x} \in (\{0,1\}^*)^n, z \in \{0,1\}^*} \equiv \left\{\mathrm{IDEAL}_{F_{mult},\mathcal{S}_1(z'),I}(\vec{x})\right\}_{\vec{x} \in (\{0,1\}^*)^n, z \in \{0,1\}^*}$$

where z' contains the same z as $\mathcal{A}$ receives, together with the $f_a(\alpha_j)$, $f_b(\alpha_j)$ values for every $j \notin I$. In order to see that the above holds, observe that both $\mathcal{S}$ and $\mathcal{S}_1$ can work when given the points of the inputs shares $\{(A'_j(\alpha_i), B'_j(\alpha_i))\}_{i \in I, j \notin I}$ and the outputs shares $\{C'_j(\alpha_i)\}_{i \in I; j \notin I}$ and they don't actually need the polynomials themselves.

Furthermore, the only difference between $\mathcal{S}$ and $\mathcal{S}_1$ is whether these polynomials are chosen with zero constant terms, or with the "correct" ones. That is, there exists a machine $\mathcal{M}$ that receives points $\{A'_j(\alpha_i), B'_j(\alpha_i)_{i\in I;j\notin I}, \{C'_j(\alpha_i)\}_{i\in I;j\notin I}$ and runs the simulation strategy with $\mathcal{A}$ while interacting with F_{mult} in an ideal execution, such that:

- If $A'_j(0) = B'_j(0) = C'_j(0) = 0$ then the joint output of $\mathcal{M}$ and the honest parties in the ideal execution is exactly that of $\text{IDEAL}_{F_{mult},\mathcal{S}(z),I}(\vec{x})$; i.e., an ideal execution with the original simulator.
- If $A'_j(0) = f_a(\alpha_j)$, $B'_j(0) = f_b(\alpha_j)$ and $C'_j(0) = f_a(\alpha_j)\cdot f_b(\alpha_j)$ then the joint output of $\mathcal{M}$ and the honest parties in the ideal execution is exactly that of $\text{IDEAL}_{F_{mult},\mathcal{S}_1(z'),I}(\vec{x})$; i.e., an ideal execution with the fictitious simulator $\mathcal{S}_1$.

By Claim 3.3, the points $\{A'_j(\alpha_i), B'_j(\alpha_i), C'_j(\alpha_i)\}_{i\in I;j\notin I}$ when $A'_j(0) = B'_j(0) = C'_j(0) = 0$ are identically distributed to the points $\{A'_j(\alpha_i), B'_j(\alpha_i), C'_j(\alpha_i)\}_{i\in I;j\notin I}$ when $A'_j(0) = f_a(\alpha_j)$, $B'_j(0) = f_b(\alpha_j)$ and $C'_j(0) = f_a(\alpha_j)\cdot f_b(\alpha_j)$. Thus, the joint outputs of the adversary and honest parties in both simulations are identical.

The Fictitious Simulator $\mathcal{S}_2$ Let $\mathcal{S}_2$ be exactly the same as $\mathcal{S}_1$, except that instead of computing $\vec{Y}_A(x)$ and $\vec{Y}_B(x)$ via the error vectors $(e_1^A,\ldots,e_n^A)$ and $(e_1^B,\ldots,e_n^B)$, it computes them like in a real execution. Specifically, it uses the actual polynomials $A_1(x),\ldots,A_n(x)$; observe that $\mathcal{S}_2$ has these polynomials since it chose them.[14] The fact that

$$\left\{\text{IDEAL}_{F_{mult},\mathcal{S}_2(z'),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{F_{mult},\mathcal{S}_1(z'),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*}$$

follows from exactly the same argument as in F_{eval} regarding the construction of the vector of polynomials $\vec{Y}(x)$, using the special property of the syndrome function.

An Ideal Execution with $\mathcal{S}_2$ and a Real Protocol Execution It remains to show that the joint outputs of the adversary and honest parties are identical in a real protocol execution and in an ideal execution with $\mathcal{S}_2$:

$$\left\{\text{HYBRID}_{\pi,\mathcal{A}(z),I}^{F_{VSS}^{subshare},F_{VSS}^{mult}}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{F_{mult},\mathcal{S}_2(z'),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*}.$$

The only difference between these two executions is the way the polynomial defining the output is chosen. Recall that in an ideal execution the output shares are generated by F_{mult} choosing a random degree-t polynomial $f_{ab}(x)$ under the constraints that $f_{ab}(0) = f_a(0)\cdot f_b(0)$, and $f_{ab}(\alpha_i) = \delta_i$ for every $i\in I$. In contrast, in a real execution,

[14] We remark that the original $\mathcal{S}$ could not work in this way since our proof that the simulations by $\mathcal{S}$ and $\mathcal{S}_1$ are identical uses the fact that the points $\{A'_j(\alpha_i), B'_j(\alpha_i)_{i\in I;j\notin I}, \{C'_j(\alpha_i)\}_{i\in I;j\notin I}$ alone suffice for simulation. This is true when computing $\vec{Y}_A(x)$ and $\vec{Y}_B(x)$ via the error vectors, but not when computing them from the actual polynomials as $\mathcal{S}_2$ does.

the output shares are derived from the polynomial $Q(x) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(x)$. However, by the way that $\mathcal{S}_2$ is defined, we have that each $\delta_i = Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(\alpha_i)$ where all polynomials $C'_1(x), \ldots, C'_n(x)$ are chosen with the correct constant terms. Thus, it remains to show that the following distributions are identical:

- *Ideal with* $\mathcal{S}_2$: Choose a degree-t polynomial $f_{ab}(x)$ at random under the constraints that $f_{ab}(0) = f_a(0) \cdot f_b(0)$, and $f_{ab}(\alpha_i) = Q(\alpha_i) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(\alpha_i)$ for every $i \in I$.
- *Real execution*: Compute $f_{ab}(x) = Q(x) = \sum_{\ell=1}^{n} \lambda_\ell \cdot C'_\ell(x)$.

We stress that in both cases, the polynomials $C'_1(x), \ldots, C'_n(x)$ have exactly the same distribution.

Observe that if $|I| = t$, then the constraints in the ideal execution with $\mathcal{S}_2$ fully define $f_{ab}(x)$ to be exactly the same polynomial as in the real execution (this is due to the fact that the constraints define $t+1$ points on a degree-t polynomial).

If $|I| < t$, then the polynomial $f_{ab}(x)$ in the ideal execution with $\mathcal{S}_2$ can be chosen by choosing $t - |I|$ random values $\beta_\ell \in_R \mathbb{F}$ (for $\ell \notin I$) and letting $f_{ab}(x)$ be the unique polynomial fulfilling the given constraints and passing through the points $(\alpha_\ell, \beta_\ell)$. Consider now the polynomial $f_{ab}(x)$ generated in a real execution. Fix any $j \notin I$. By the way that Protocol 6.17 works, $C'_j(x)$ is a random polynomial under the constraint that $C'_j(0) = f_a(\alpha_j) \cdot f_b(\alpha_j)$. By Corollary 3.2, given points $\{(\alpha_i, C'_j(\alpha_i))\}_{i \in I}$ and a "secret" $s = C'_j(0)$, it holds that any subset of $t - |I|$ points of $\{C'_j(\alpha_\ell)\}_{\ell \notin I}$ are *uniformly distributed* (note that none of the points in $\{C'_j(\alpha_\ell)\}_{\ell \notin I}$ are seen by the adversary). This implies that for any $t - |I|$ points α_ℓ (with $\ell \notin I$) the points $f_{ab}(\alpha_\ell)$ in the polynomial $f_{ab}(x)$ computed in a real execution are uniformly distributed. This is therefore exactly the same as choosing $t - |I|$ values $\beta_\ell \in_R \mathbb{F}$ at random (with $\ell \notin I$), and setting f_{ab} to be the unique polynomial such that $f_{ab}(\alpha_\ell) = \beta_\ell$ in addition to the above constraints. Thus, the polynomials $f_{ab}(x)$ computed in an ideal execution with $\mathcal{S}_2$ and in a real execution are identically distributed. This implies that the $\text{HYBRID}_{\pi,\mathcal{A}(z),I}^{F_{VSS}^{subshare}, F_{VSS}^{mult}}(\vec{x})$ and $\text{IDEAL}_{F_{mult},\mathcal{S}_2(z'),I}(\vec{x})$ distributions are identical, as required. □

Securely Computing F_{mult} in the Plain Model The following corollary is obtained by combining the following:

- Theorem 5.7 (securely compute F_{VSS} in the plain model),
- Theorem 6.6 (securely compute F_{mat}^{A} in the F_{VSS}-hybrid model),
- Theorem 6.9 (securely compute $F_{VSS}^{subshare}$ in the F_{mat}^{A}-hybrid model),
- Theorem 6.12 (securely compute F_{eval} in the $F_{VSS}^{subshare}$-hybrid model),
- Theorem 6.15 (securely compute F_{VSS}^{mult} in the F_{VSS}, F_{eval}-hybrid model), and
- Theorem 6.18 (securely compute F_{mult} in the $F_{VSS}^{subshare}$, F_{VSS}^{mult}-hybrid model)

and using the modular sequential composition theorem of [8]. We have:

Corollary 6.19. *Let $t < n/3$. Then, there exists a protocol that is t-secure for F_{mult} functionality in the plain model with private channels, in the presence of a static malicious adversary.*

More Efficient Constant-Round Multiplication [2]. The protocol that we have presented is very close to that described by BGW. However, it is possible to use these techniques to achieve a more efficient multiplication protocol. For example, observe that if the parties already hold shares of all other parties' shares, then these can be used directly in F_{VSS}^{mult} without running $F_{VSS}^{subshare}$ at all. Now, the verifiable secret-sharing protocol of Ben-Or et al. [7] presented in Sect. 5 is based on bivariate polynomials, and so all parties do indeed receive shares of all other parties' shares. This means that it is possible to modify Protocol 6.17 so that the parties proceed directly to F_{VSS}^{mult} without using $F_{VSS}^{subshare}$ at all. Furthermore, the output of each party P_i in F_{VSS}^{mult} is the share $C(\alpha_i)$ received via the F_{VSS} functionality; see Protocol 6.14. Once again, using VSS based on bivariate polynomials, this means that the parties can actually output the shares of all other parties' shares as well. Applying the linear computation of $Q(x)$ to these bivariate shares, we conclude that it is possible to include the shares of all other parties as additional output from Protocol 6.17. Thus, the next time that F_{mult} is called, the parties will again already have the shares of all other parties' shares and $F_{VSS}^{subshare}$ need not be called. This is a significant efficiency improvement. (Note that unless some of the parties behave maliciously, F_{VSS}^{mult} itself requires $t+1$ invocations of F_{VSS} and nothing else. With this efficiency improvement, we have that the entire cost of F_{mult} is $n \cdot (t+1)$ invocations of F_{VSS}.) See [2] for more details on this and other ways to further utilize the properties of bivariate secret sharing in order to obtain simpler and much more efficient multiplication protocols.

We remark that there exist protocols that are *not* constant round and have far more efficient communication complexity; see [5] for such a protocol. In addition, in the case of $t < n/4$, there is a much more efficient solution for constant-round multiplication presented in BGW itself; see "Appendix" for a brief description.

7. Secure Computation in the (F_{VSS}, F_{mult})-Hybrid Model

7.1. *Securely Computing any Functionality*

In this section we show how to t-securely compute any functionality f in the (F_{VSS}, F_{mult})-hybrid model, in the presence of a malicious adversary controlling any $t < n/3$ parties. We also assume that all inputs are in a known field $\mathbb{F}$ (with $|\mathbb{F}| > n$), and that the parties all have an arithmetic circuit C over $\mathbb{F}$ that computes f. As in the semi-honest case, we assume that $f\colon \mathbb{F}^n \to \mathbb{F}^n$ and so the input and output of each party is a single field element.

The protocol here is almost identical to Protocol 4.1 for the semi-honest case; the only difference is that the verifiable secret-sharing functionality F_{VSS} is used in the input stage, and the F_{mult} functionality used for multiplication gates in the computation stage is the corruption-aware one defined for the case of malicious adversaries (see Sect. 6.7). See Sect. 5.4 for the definition of F_{VSS} (Functionality 5.5), and see Functionality 6.16 for the definition of F_{mult}. Observe that the definition of F_{VSS} is such that the effect is identical to that of Shamir secret sharing in the presence of semi-honest adversaries. Furthermore, the correctness of F_{mult} ensures that at every intermediate stage the (honest) parties hold correct shares on the wires of the circuit. In addition, observe that F_{mult} reveals nothing to the adversary except for its points on the input wires, which it already knows. Thus,

PROTOCOL 7.1 (t-Secure Computation of f in the (F_{mult}, F_{VSS})-Hybrid Model).

- **Inputs**: Each party P_i has an input $x_i \in \mathbb{F}$.
- **Common input**: Each party P_i holds an arithmetic circuit C over a field $\mathbb{F}$ of size greater than n, such that for every $\vec{x} \in \mathbb{F}^n$ it holds that $C(\vec{x}) = f(\vec{x})$, where $f\colon \mathbb{F}^n \to \mathbb{F}^n$. The parties also hold a description of $\mathbb{F}$ and distinct nonzero values $\alpha_1, \ldots, \alpha_n$ in $\mathbb{F}$.
- **Aiding ideal functionality initialization**: Upon invocation, the trusted parties computing the (fictitiously corruption-aware) functionality F_{VSS} and the corruption-aware functionality F_{mult} receive the set of corrupted parties I.
- **The protocol**:
  1. **The input-sharing stage**:
     (a) Each party P_i chooses a polynomial $q_i(x)$ uniformly at random from the set $\mathcal{P}^{x_i,t}$ of degree-t polynomials with constant term x_i. Then, P_i invokes the F_{VSS} functionality as dealer, using $q_i(x)$ as its input.
     (b) Each party P_i records the values $q_1(\alpha_i), \ldots, q_n(\alpha_i)$ that it received from the F_{VSS} functionality invocations. If the output from F_{VSS} is $\perp$ for any of these values, P_i replaces the value with 0.
  2. **The circuit emulation stage**: Let $G_1, \ldots, G_\ell$ be a predetermined topological ordering of the gates of the circuit. For $k = 1, \ldots, \ell$ the parties work as follows:
     - *Case 1—G_k is an addition gate*: Let β_i^k and γ_i^k be the shares of input wires held by party P_i. Then, P_i defines its share of the output wire to be $\delta_i^k = \beta_i^k + \gamma_i^k$.
     - *Case 2—G_k is a multiplication-by-a-constant gate with constant c*: Let β_i^k be the share of the input wire held by party P_i. Then, P_i defines its share of the output wire to be $\delta_i^k = c \cdot \beta_i^k$.
     - *Case 3—G_k is a multiplication gate*: Let β_i^k and γ_i^k be the shares of input wires held by party P_i. Then, P_i sends (β_i^k, γ_i^k) to the ideal functionality F_{mult} and receives back a value δ_i^k. Party P_i defines its share of the output wire to be δ_i^k.
  3. **The output reconstruction stage**:
     (a) Let $o_1, \ldots, o_n$ be the output wires, where party P_i's output is the value on wire o_i. For every $i = 1, \ldots, n$, denote by $\beta_1^i, \ldots, \beta_n^i$ the shares that the parties hold for wire o_i. Then, each P_j sends P_i the share β_j^i.
     (b) Upon receiving all shares, P_i runs the Reed–Solomon decoding procedure on the possible corrupted codeword $(\beta_1^i, \ldots, \beta_n^i)$ to obtain a codeword $(\tilde{\beta}_1^i, \ldots, \tilde{\beta}_n^i)$. Then, P_i computes $\mathsf{reconstruct}_{\vec{\alpha}}(\tilde{\beta}_1^i, \ldots, \tilde{\beta}_n^i)$ and obtains a polynomial $g_i(x)$. Finally, P_i then defines its output to be $g_i(0)$.

the adversary learns nothing in the computation stage, and after this stage the parties all hold correct shares on the circuit-output wires. The protocol is therefore concluded by having the parties send their shares on the output wires to the appropriate recipients (i.e., if party P_j is supposed to receive the output on a certain wire, then all parties send their shares on that wire to P_j). This step introduces a difficulty that does not arise in the semi-honest setting; some of the parties may send *incorrect* values on these wires. Nevertheless, as we have seen, this can be easily solved since it is guaranteed that more than two-thirds of the shares are correct and so each party can apply Reed–Solomon decoding to ensure that the final output obtained is correct. See Protocol 7.1 for full details.

We now prove that Protocol 7.1 can be used to securely compute any functionality. We stress that the theorem holds for regular functionalities only, and not for corruption-aware functionalities (see Sect. 6.2). This is because not every corruption-aware functionality can be computed by a circuit that receives inputs from the parties only, without having the set of identities of the corrupted parties as auxiliary input (such a circuit is what is needed for Protocol 7.1).

Theorem 7.2. *Let $f\colon \mathbb{F}^n \to \mathbb{F}^n$ be any n-ary functionality, and let $t < n/3$. Then, Protocol 7.1 (with auxiliary input C to all parties) is t-secure for f in the (F_{VSS}, F_{mult})-hybrid model, in the presence of a static malicious adversary.*

Proof. Intuitively, security here follows from the fact that a corrupted party in Protocol 7.1 cannot do anything but choose its input as it wishes. In order to see this, observe that the entire protocol is comprised of F_{VSS} and F_{mult} calls, and in the latter the adversary receives no information in its output and has no influence whatsoever on the outputs of the honest parties. Finally, the adversary cannot affect the outputs of the honest parties due to the Reed–Solomon decoding carried out in the output stage. The simulator internally invokes $\mathcal{A}$ and simulates the honest parties in the protocol executions and the invocations of F_{VSS} and F_{mult} functionalities and externally interacts with the trusted party computing f. We now formally describe the simulator.

The Simulator $\mathcal{S}$

- *$\mathcal{S}$ internally invokes $\mathcal{A}$ with its auxiliary input z.*
- **The input-sharing stage**:
  1. *For every $j \notin I$, $\mathcal{S}$ chooses a uniformly distributed polynomial $q_j(x) \in_R \mathcal{P}^{0,t}$ (i.e., degree-t polynomial with constant term 0), and for every $i \in I$, it internally sends the adversary $\mathcal{A}$ the shares $q_j(\alpha_i)$ as it expects from the F_{VSS} invocations.*
  2. *For every $i \in I$, $\mathcal{S}$ internally obtains from $\mathcal{A}$ the polynomial $q_i(x)$ that it instructs P_i to send to the F_{VSS} functionality when P_i is the dealer. If $\deg(q_i(x)) \leq t$, $\mathcal{S}$ simulates F_{VSS} sending $q_i(\alpha_\ell)$ to P_ℓ for every $\ell \in I$. Otherwise, $\mathcal{S}$ simulates F_{VSS} sending $\perp$ to P_ℓ for every $\ell \in I$, and resets $q_i(x)$ to be a constant polynomial equaling zero everywhere.*
  3. *For every $j \in \{1, \ldots, n\}$, denote the circuit-input wire that receives P_j's input by w_j. Then, for every $i \in I$, simulator $\mathcal{S}$ stores the value $q_j(\alpha_i)$ as the share of P_i on the wire w_j.*
- **Interaction with the trusted party**:
  1. *$\mathcal{S}$ externally sends the trusted party computing f the values $\{x_i = q_i(0)\}_{i \in I}$ as the inputs of the corrupted parties.*
  2. *$\mathcal{S}$ receives from the trusted party the outputs $\{y_i\}_{i \in I}$ of the corrupted parties.*
- **The circuit emulation stage**: *Let $G_1, \ldots, G_\ell$ be the gates of the circuit according to their topological ordering. For $k = 1, \ldots, \ell$:*
  1. Case 1—G_k is an addition gate: *Let β_i^k and γ_i^k be the shares that $\mathcal{S}$ has stored for the input wires to G_k for the party P_i. Then, for every $i \in I$, $\mathcal{S}$ computes*

the value $\delta_i^k = \beta_i^k + \gamma_i^k$ *as the share of* P_i *for the output wire of* G_k *and stores this values.*

2. Case 2—G_k is a multiplication-by-a-constant gate with constant c: *Let* β_i^k *be the share that* $\mathcal{S}$ *has stored for the input wire to* G_k *for* P_i*. Then, for every* $i \in I$, $\mathcal{S}$ *computes the value* $\delta_i^k = c \cdot \beta_i^k$ *as the share of* P_i *for the output wire of* G_k *and stores this value.*
3. Case 3—G_k is a multiplication gate: $\mathcal{S}$ *internally simulates the trusted party computing* F_{mult} *for* $\mathcal{A}$*, as follows. Let* β_i^k *and* γ_i^k *be the shares that* $\mathcal{S}$ *has stored for the input wires to* G_k *for the party* P_i*. Then,* $\mathcal{S}$ *first hands* $\{(\beta_i^k, \gamma_i^k)\}_{i \in I}$ to $\mathcal{A}$ *as if coming from* F_{mult} (*see Step 2 of Functionality* 6.16) *Next, it obtains from* $\mathcal{A}$ *values* $\{\delta_i^k\}_{i \in I}$ *as the input of the corrupted parties for the functionality* F_{mult} *(See step 3 of Functionality* 6.16*). If any* δ_i^k *is not sent, then* $\mathcal{S}$ *sets* $\delta_i^k = 0$*. Finally,* $\mathcal{S}$ *stores* δ_i^k *as the share of* P_i *for the output wire of* G_k*. (Note that the adversary has no output from* F_{mult} *beyond receiving its own* (β_i^k, γ_i^k) *values.)*

- **The output reconstruction stage**: *For every* $i \in I$*, simulator* $\mathcal{S}$ *works as follows. Denote by* o_i *the circuit-output wire that contains the output of party* P_i*, and let* $\{\beta_\ell^i\}_{\ell \in I}$ *be the shares that* $\mathcal{S}$ *has stored for wire* o_i *for all corrupted parties* P_ℓ $(\ell \in I)$*. Then,* $\mathcal{S}$ *chooses a random polynomial* $q_i'(x)$ *under the constraint that* $q_i'(\alpha_\ell) = \beta_\ell^i$ *for all* $\ell \in I$*, and* $q_i'(0) = y_i$*, where* y_i *is the output of* P_i *received by* $\mathcal{S}$ *from the trusted party computing* f*. Finally, for every* $j \notin I$, $\mathcal{S}$ *simulates the honest party* P_j *sending* $q_i'(\alpha_j)$ *to* P_i*.*

A Fictitious Simulator $\mathcal{S}'$ We begin by constructing a fictitious simulator $\mathcal{S}'$ that works exactly like $\mathcal{S}$ except that it receives as input all of the input values $\vec{x} = (x_1, \ldots, x_n)$, and chooses the polynomials $q_j(x) \in_R \mathcal{P}^{x_j,t}$ of the honest parties with the correct constant term instead of with constant term 0. Apart from this, $\mathcal{S}'$ works exactly like $\mathcal{S}$ and interacts with a trusted party computing f in the ideal model.

The Original and Fictitious Simulations We now show that the joint output of the adversary and honest parties is identical in the original and fictitious simulations. That is,

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{f,\mathcal{S}'(\vec{x},z),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n, z\in\{0,1\}^*}. \quad (7.1)$$

This follows immediately from the fact that both $\mathcal{S}$ and $\mathcal{S}'$ can work identically when receiving the points $\{q_j(\alpha_i)\}_{i \in I; j \notin I}$ externally. Furthermore, the only difference between them is if $q_j(\alpha_i) \in_R \mathcal{P}^{0,t}$ or $q_j(\alpha_i) \in_R \mathcal{P}^{x_j,t}$, for every $j \notin I$. Thus, there exists a single machine $\mathcal{M}$ that runs in the ideal model with a trusted party computing f, and that receives points $\{q_j(\alpha_i)\}_{i \in I; j \notin I}$ and runs the simulation using these points. Observe that if $q_j(\alpha_i) \in_R \mathcal{P}^{0,t}$ for every $j \notin I$, then the joint output of $\mathcal{M}$ and the honest parties in the ideal execution is exactly the same as in the ideal execution with $\mathcal{S}$. In contrast, if $q_j(\alpha_i) \in_R \mathcal{P}^{x_j,t}$ for every $j \notin I$, then the joint output of $\mathcal{M}$ and thehonest parties in the ideal execution is exactly the same as in the ideal execution with the fictitious simulator $\mathcal{S}'$. By Claim 3.3, these points are identically distributed in both cases, and thus the joint output of $\mathcal{M}$ and the honest parties is identically distributed in both cases; Eq. (7.1) follows.

The Fictitious Simulation and a Protocol Execution We now proceed to show that:

$$\left\{\text{IDEAL}_{f,\mathcal{S}'(\vec{x},z),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*} \equiv \left\{\text{HYBRID}^{F_{VSS},F_{mult}}_{\pi,\mathcal{A}(z),I}(\vec{x})\right\}_{\vec{x}\in(\{0,1\}^*)^n,z\in\{0,1\}^*}.$$

We first claim that the output of the honest parties is identically distributed in the real execution and the alternative simulation. This follows immediately from the fact that the inputs to F_{VSS} fully determine the inputs $\vec{x}$, which in turn fully determine the output of the circuit. In order to see this, observe that F_{mult} always sends shares of the product of the input shares (this holds as long as the honest parties send "correct" inputs which they always do), and the local computation in the case of multiplication-by-a-constant and addition gates is trivially correct. Thus, the honest parties all hold correct shares of the outputs on the circuit-output wires. Finally, by the Reed–Solomon decoding procedure (with code length n and dimension $t+1$), it is possible to correct up to $\frac{n-t}{2} > \frac{3t-t}{2} = t$ errors. Thus, the values sent by the corrupted parties in the output stage have no influence whatsoever on the honest parties' outputs.

Next, we show that the view of the adversary $\mathcal{A}$ in the fictitious simulation with $\mathcal{S}'$ is identical to its view in real protocol execution, conditioned on the honest parties' outputs $\{y_j\}_{j\notin I}$. It is immediate that these views are identical up to the output stage. This is because $\mathcal{S}'$ uses the same polynomials as the honest parties in the input stage, and in the computation stage $\mathcal{A}$ receives no output at all (except for its values on the input wires for multiplication gates which are already known). It thus remains to show that the values $\{q'_i(\alpha_j)\}_{i\in I;j\notin I}$ received by $\mathcal{A}$ from $\mathcal{S}'$ in the output stage are identically distributed to the values received by $\mathcal{A}$ from the honest parties P_j.

Assume for simplicity that the output wire comes directly from a multiplication gate. Then, F_{mult} chooses the polynomial that determines the shares on the wire at random, under the constraint that it has the correct constant term (which in this case we know is y_i, since we have already shown that the honest parties' outputs are correct). Since this is *exactly* how $\mathcal{S}'$ chooses the value, we have that the distributions are identical. This concludes the proof. □

Putting It All Together We conclude with a corollary that considers the plain model with private channels. The corollary is obtained by combining Theorem 5.7 (securely computing F_{VSS} in the plain model), Corollary 6.19 (securely computing F_{mult} in the plain model) and Theorem 7.2 (securely computing f in the F_{VSS}, F_{mult}-hybrid model), and using the modular sequential composition theorem of [8]:

Corollary 7.3. *For every functionality $f\colon \mathbb{F}^n \to \mathbb{F}^n$ and $t < n/3$, there exists a protocol that is t-secure for f in the plain model with private channels, in the presence of a static malicious adversary.*

7.2. *Communication and Round Complexity*

We begin by summarizing the *communication complexity* of the BGW protocol (as presented here) in the case of malicious adversaries. We consider both the cost in the "optimistic case" where no party deviates from the protocol specification, and in the "pessimistic case" where some party does deviate. We remark that since the protocol achieves perfect security, nothing can be gained by deviating, except possible to make the parties run longer. Thus, in general, one would expect that the typical cost of running the protocol is the "optimistic cost." In addition, we separately count the number of field elements sent over the point-to-point private channels, and the number of elements sent over a broadcast channel. (The "BGW" row in the table counts the overall cost of computing a circuit C with $|C|$ multiplication gates.)

Protocol	Optimistic cost	Pessimistic cost
F_{VSS}	$O(n^2)$ over pt-2-pt	$O(n^2)$ over pt-2-pt
	No broadcast	$O(n^2)$ broadcast
$F_{VSS}^{subshare}$	$O(n^3)$ over pt-2-pt	$O(n^3)$ over pt-2-pt
	No broadcast	$O(n^3)$ broadcast
F_{eval}	$O(n^3)$ over pt-2-pt	$O(n^3)$ over pt-2-pt
	No broadcast	$O(n^3)$ broadcast
F_{VSS}^{mult}	$O(n^3)$ over pt-2-pt	$O(n^5)$ over pt-2-pt
	No broadcast	$O(n^5)$ broadcast
F_{mult}	$O(n^4)$ over pt-2-pt	$O(n^6)$ over pt-2-pt
	No broadcast	$O(n^6)$ broadcast
BGW	$O(\|C\| \cdot n^4)$ over pt-2-pt	$O(\|C\| \cdot n^6)$ over pt-2-pt
	No broadcast	$O(\|C\| \cdot n^6)$ broadcast

Regarding *round complexity*, since we use the sequential composition theorem, all calls to functionalities must be sequential. However, in Sect. 8 we will see that all subprotocols can actually be run concurrently, and thus in parallel. In this case, we have that all the protocols for computing F_{VSS}, $F_{VSS}^{subshare}$, F_{eval}, F_{VSS}^{mult} and F_{mult} have a *constant number of rounds*. Thus, each level of the circuit C can be computed in $O(1)$ rounds, and the overall round complexity is linear in the depth of the circuit C. This establishes the complexity bounds stated in Theorem 1.

8. Adaptive Security, Composition and the Computational Setting

Our proof of the security of the BGW protocol in the semi-honest and malicious cases relates to the *stand-alone model* and to the case of *static corruptions*. In addition, in the information-theoretic setting, we consider perfectly secure private channels. In this section, we show that our proof of security for the limited stand-alone model with static corruptions suffices for obtaining security in the much more complex settings of composition and adaptive corruptions (where the latter is for a weaker variant; see below). This is made possible due to the fact that the BGW protocol is *perfectly secure*, and not just statistically secure.

Security Under Composition In [24, Theorem3] it was proven that any protocol that computes a functionality f with perfect security and has a straight-line black-box simulator (as is the case with all of our simulators), securely computes f under the definition of (static) universal composability [9] (or equivalently, concurrent general composition [26]). Using the terminology UC-secure to mean secure under the definition of universal composability, we have the following corollary:

Corollary 8.1. *For every functionality f, there exists a protocol for UC-securely computing f in the presence of* static *semi-honest adversaries that corrupt up to $t < n/2$ parties, in the private channels model. Furthermore, there exists a protocol for UC-securely computing f in the presence of* static *malicious adversaries that corrupt up to $t < n/3$ parties, in the private channels model.*

Composition in the Computational Setting There are two differences between the information-theoretic and computational settings. First, in the information-theoretic setting there are ideally private channels, whereas in the computational setting it is typically only assumed that there are authenticated channels. Second, in the information-theoretic setting, the adversary does not necessarily run in polynomial time. Nevertheless, as advocated by [20, Sec. 7.6.1] and adopted in Definition 2.3, we consider simulators that run in time that is polynomial in the running time of the adversary. Thus, if the real adversary runs in polynomial time, then so does the simulator, as required for the computational setting. This is also means that it is possible to replace the ideally private channels with public-key encryption. We state our corollary here for computational security for the most general setting of UC-security (although an analogous corollary can of course be obtained for the more restricted stand-alone model as well). The corollary is obtained by replacing the private channels in Corollary 8.1 with UC-secure channels that can be constructed using semantically secure public-key encryption [9,12]. We state the corollary only for the case of malicious adversaries since the case of semi-honest adversaries has already been proven in [13] for any $t < n$.

Corollary 8.2. *Assuming the existence of semantically secure public-key encryption, for every functionality f, there exists a protocol for UC-securely computing f in the presence of* static *malicious adversaries that corrupt up to $t < n/3$ parties, in the authenticated channels model.*

We stress that the above protocol requires no common reference string or other setup (beyond that required for obtaining authenticated channels). This is the first full proof of the existence of such a UC-secure protocol.

Adaptive Security with Inefficient Simulation In general, security in the presence of a static adversary does not imply security in the presence of an adaptive adversary, even for perfectly secure protocols [10]. This is true, for example, for the definition of security of adaptive adversaries that appears in [8]. However, there is an alternative definition of security (for static and adaptive adversaries) due to Dodis and Micali [16] that requires a straight-line black-box simulator, and also the existence of a committal round at which point the transcript of the protocol fully defines all of the parties' inputs. Furthermore, it was shown in [10] that security in the presence of static adversaries in the strong sense of Dodis and Micali [16] *does* imply security in the presence of adaptive adversaries (also in the strong sense of Dodis and Micali [16]), as long as the simulator is allowed to be inefficient (i.e., the simulator is not required to be of *comparable complexity* to the adversary; see Definition 2.3). It turns out that all of the protocols in this paper meet this definition. Thus, applying the result of Canetti et al. [10] we can conclude that all of the protocols in this paper are secure in the presence of adaptive adversaries with inefficient simulation, under the definition of Dodis and Micali [16]. Finally, we observe that any protocol that is secure in the presence of adaptive adversaries under the definition of Dodis and Micali [16] is also secure in the presence of adaptive adversaries under the definition of Canetti [8]. We therefore obtain security in the presence of adaptive adversaries with *inefficient simulation* "for free." This is summarized as follows.

Corollary 8.3. *For every functionality f, there exists a protocol for securely computing f in the presence of adaptive semi-honest adversaries that corrupt up to $t < n/2$ parties with, in the private channels model (with inefficient simulation). Furthermore, there exists a protocol for securely computing f in the presence of adaptive malicious adversaries that corrupt up to $t < n/3$ parties, in the private channels model (with inefficient simulation).*

Acknowledgements

We thank Tal Rabin and Ivan Damgård for helpful discussions. We are deeply in gratitude to Oded Goldreich for the significant time and effort that he dedicated to helping us in this work.

9. Appendix: Multiplication in the Case of $t < n/4$

In this section, we describe how to securely compute shares of the product of shared values, in the presence of a malicious adversary controlling only $t < n/4$ parties. This is much simpler than the case of $t < n/3$, since in this case there is enough redundancy to correct errors in polynomials with degree-$2t$. Due to this, it is similar in spirit to the semi-honest multiplication protocol, using the simplification of Gennaro et al. [19]. In this appendix, we provide a full description of this simpler and more efficient protocol, without a proof of security. In our presentation here, we assume familiarity with the material appearing in Sects. 6.2, 6.3, 6.4 and 6.7.

High-Level Description of the Protocol Recall that the multiplication protocol works by having the parties compute a linear function of the product of their shares. That is, each party locally multiplies its two shares, and then subshares the result using a degree-t polynomial. The final result is then a specific linear combination of these subshares. Similarly to the case of $t < n/3$ we need a mechanism that verifies that the corrupted parties have shared the correct products. In this case where $t < n/4$, this can be achieved by directly using the error correction property of the Reed–Solomon code, since we can correct degree-$2t$ polynomials.

The high-level protocol is as follows:

- Each party holds inputs a_i and b_i, which are shares of two degree-t polynomials that hide values a and b, respectively.
- Each party locally computes the product $a_i \cdot b_i$. The parties then distribute subshares of $a_i \cdot b_i$ to all other parties in a verifiable way using a variant of the $F_{VSS}^{subshare}$. Observe that the products are points on degree-$2t$ polynomials. Thus, these shares constitute a Reed–Solomon code with parameters $[4t+1, 2t+1, 2t+1]$ for which it is possible to correct up to t errors. There is therefore enough redundancy to correct errors, unlike the case where $t < n/3$ where t errors can not necessarily be corrected on a $2t$-degree polynomial. This enables us to design a variant of the $F_{VSS}^{subshare}$ functionality (Sect. 6.4) that works directly on the products $a_i \cdot b_i$.
- At this point, all parties verifiably hold (degree-t) subshares of the product of the input shares of every party. As shown in [19], shares of the product of the values on the wires can be obtained by computing a linear function of the subshares obtained in the previous step.

In the following, we show how to slightly modify the $F_{VSS}^{subshare}$ *functionality* (Sect. 6.4) to work with the case of $t < n/4$ (as we will explain, the protocol actually remains the same). In addition, we provide a full specification for the protocol that implements the multiplication functionality, F_{mult}; i.e., the modifications to Protocol 6.17.

We stress that in the case that $t < n/3$ it is not possible to run $F_{VSS}^{subshare}$ directly on the products $a_i \cdot b_i$ of the input shares since they define a degree-$2t$ polynomial and so at most $\frac{n-2t-1}{2} = t/2$ errors can be corrected. Thus, it is necessary to run $F_{VSS}^{subshare}$ separately on a_i and b_i, and then use the F_{mult} functionality to achieve a sharing of $a_i \cdot b_i$. It follows that in this case of $t < n/4$, there is no need for the involved F_{VSS}^{mult} functionality, making the protocol simpler and more efficient.

The $F_{VSS}^{subshare}$ Functionality and Protocol We reconsider the definition of the $F_{VSS}^{subshare}$ functionality, and present the necessary modifications for the functionality. Here, we assume that the inputs of the $3t+1$ honest parties $\{(\alpha_j, \beta_j)\}_{j \notin I}$ define a degree-$2t$ polynomial instead of a degree-t polynomial. The definition of the functionality remains unchanged except for this modification.

We now proceed to show that Protocol 6.8 that implements the $F_{VSS}^{subshare}$ functionality works as is also for this case, where the inputs are shares of a degree-$2t$ polynomial. In order to see this, recall that there are two steps in the protocol that may be affected by the change of the inputs and should be reconsidered: (1) the parity-check matrix H, which is the parameter for the F_{mat}^{H}-functionality, and (2) Step 3, where each party locally computed the error vector using the syndrome vector (the output of the F_{mat}^{H}), and the error correction procedure of the Reed–Solomon code. These steps could conceivably be different since in this case the parameters of the Reed–Solomon codes are different. Regarding the parity-check matrix, the same matrix is used for both cases. Recall that the case of $t < n/3$ defines a Reed–Solomon code with parameters $[3t+1, t+1, 2t+1]$, and the case of $t < n/4$ defines a code with parameters $[4t+1, 2t+1, 2t+1]$. Moreover, recall that a Reed–Solomon code with parameters $[n, k, n-k+1]$ has a parity-check matrix $H \in \mathbb{F}^{(n-k)\times n}$. In the case of $n = 3t+1$ we have that $k = t+1$ and so $n-k = 2t$. Likewise, in the case of $n = 4t+1$, we have that $k = 2t+1$ and so $n-k = 2t$. It follows that in both case, the parity-check matrix H is of dimension $2t \times n$, and so is the same (of course, for different values of t a different matrix is used, but what we mean is that the protocol description is exactly the same). Next, in Step 3 of the protocol, each party locally executes the Reed–Solomon error correction procedure given the syndrome vector that is obtained using F_{mat}^{H}. This procedure depends on the distance of the code. However, this is $2t + 1$ in both cases and so the protocol description remains exactly the same.

The Protocol for F_{mult} We now proceed to the specification of the functionality F_{mult}. As we have mentioned, this protocol is much simpler than Protocol 6.17 since the parties can run the $F_{VSS}^{subshare}$ functionality directly on the product of their inputs, instead of first running it on a_i, then on b_i, and then using F_{mult} to obtain a sharing of $a_i \cdot b_i$. The protocol is as follows:

PROTOCOL 9.1 (Computing F_{mult} in the $F_{VSS}^{subshare}$-hybrid model (with $t < n/4$)).

- **Input**: Each party P_i holds a_i, b_i, where $a_i = f_a(\alpha_i)$, $b_i = f_b(\alpha_i)$ for some polynomials $f_a(x)$, $f_b(x)$ of degree t, which hide a, b, respectively. (If not all the points lie on a single degree-t polynomial, then no security guarantees are obtained. See Footnote 9.)
- **Common input**: A field description $\mathbb{F}$ and n distinct nonzero elements $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$.
- **The protocol**:
  1. Each party locally computes $c_i = a_i \cdot b_i$.
  2. The parties invoke the $F_{VSS}^{subshare}$ functionality with each party P_i using c_i as its private input. Each party P_i receives back shares $C_1(\alpha_i), \ldots, C_n(\alpha_i)$, and a polynomial $C_i(x)$. (Recall that for every i, the polynomial $C_i(x)$ is of degree-t and $C_i(0) = c_i = a_i \cdot b_i = f_a(\alpha_i) \cdot f_b(\alpha_i)$)
  3. Each party locally computes $Q(\alpha_i) = \sum_{j=1}^{n} \lambda_j \cdot C_j(\alpha_i)$, where $(\lambda_1, \ldots, \lambda_n)$ is the first row of the matrix $V_{\vec{\alpha}}^{-1}$ (see Sect. 6.7).
- **Output**: Each party P_i outputs $Q(\alpha_i)$.

References

[1] G. Asharov, Y. Lindell, A full proof of the BGW protocol for perfectly-secure multiparty computation. *Cryptology ePrint Archive*, Report 2011/136 (2011)

[2] G. Asharov, Y. Lindell, T. Rabin, Perfectly-secure multiplication for any $t < n/3$, in *CRYPTO 2011*. LNCS 6841 (Springer, 2011), pp. 240–258

[3] D. Beaver, Multiparty protocols tolerating half faulty processors, in *CRYPTO'89*. LNCS 435 (Springer, 1990), pp. 560–572

[4] D. Beaver, Foundations of secure interactive computing, in *CRYPTO'91*. LNCS 576 (Springer, 1991), pp. 377–391

[5] Z. Beerliová-Trubíniová, M. Hirt, Perfectly-secure MPC with linear communication complexity, in *5th TCC*. LNCS 4948 (Springer, 2008), pp. 213–230

[6] M. Ben-Or, R. El-Yaniv, Resilient-Optimal Interactive Consistency in Constant Time, in *Distributed Computing*, 16(4):249–262, (2003)

[7] M. Ben-Or, S. Goldwasser, A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation, in *The 20th STOC* (1988), pp. 1–10

[8] R. Canetti, Security and Composition of Multiparty Cryptographic Protocols. In the *Journal of Cryptology*, 13(1):143–202, (2000)

[9] R. Canetti, Universally composable security: a new paradigm for cryptographic protocols, in *The 42nd FOCS* (2001), pp. 136–145. See *Cryptology ePrint Archive: Report 2000/067* for the full version

[10] R. Canetti, I. Damgård, S. Dziembowski, Y. Ishai, T. Malkin, Adaptive versus Non-Adaptive Security of Multi-Party Protocols. In the *Journal of Cryptology* 17(3):153–207, (2004)

[11] R. Canetti, U. Feige, O. Goldreich, M. Naor, Adaptively secure multi-party computation, in *The 28th STOC* (1996), pp. 639–648

[12] R. Canetti, H. Krawczyk, Universally composable notions of key-exchange and secure channels, in *EUROCRYPT 2002*. LNCS 2332 (Springer, 2002), pp. 337–351

[13] R. Canetti, Y. Lindell, R. Ostrovsky, A. Sahai, Universally composable two-party and multi-party computation. In *The 34th STOC* (2002), pp. 494–503

[14] D. Chaum, C. Crépeau, I. Damgård, Multi-party unconditionally secure protocols, in *20th STOC* (1988), pp. 11–19

[15] B. Chor, S. Goldwasser, S. Micali, B. Awerbuch, Verifiable secret sharing and achieving simultaneity in the presence of faults, in *The 26 FOCS* (1985), pp. 383–395

[16] Y. Dodis, S. Micali, Parallel reducibility for information-theoretically secure computation, in *CRYPTO 2000*. LNCS 1880 (Springer, 2000), pp. 74–92

[17] P. Feldman, Optimal algorithms for byzantine agreement. Ph.D. thesis, Massachusetts Institute of Technology (1988)

[18] P. Feldman, S. Micali, An Optimal Probabilistic Protocol for Synchronous Byzantine Agreement. In the *SIAM Journal on Computing*, 26(4):873–933, (1997)

[19] R. Gennaro, M.O. Rabin, T. Rabin, Simplified VSS and fact-track multiparty computations with applications to threshold cryptography, in *The 17th PODC* (1998), pp. 101–111

[20] O. Goldreich, *Foundations of Cryptography: Volume 2—Basic Applications* (2004, Cambridge University Press, Cambridge)

[21] O. Goldreich, S. Micali, A. Wigderson, How to play any mental game—a completeness theorem for protocols with honest majority, in *19th STOC* (1987), pp. 218–229. For details see [20]

[22] S. Goldwasser, L. Levin, Fair computation of general functions in presence of immoral majority, in *CRYPTO'90*. LNCS 537 (Springer, 1990), pp. 77–93

[23] S. Goldwasser, S. Micali, Probabilistic Encryption. *JCSS*, 28(2):270–299, (1984)

[24] E. Kushilevitz, Y. Lindell, T. Rabin, Information-Theoretically Secure Protocols and Security Under Composition. In the *SIAM Journal on Computing*, 39(5):2090–2112, (2010)

[25] L. Lamport, R. Shostack, M. Pease, The Byzantine Generals Problem. In the *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, (1982)

[26] Y. Lindell, General composition and universal composability in secure multi-party computation, in *The 44th FOCS* (2003), pp. 394–403

[27] Y. Lindell, A. Lysyanskaya, T. Rabin, Sequential composition of protocols without simultaneous termination, in *The 21st PODC* (2002), pp. 203–212

[28] R.J. McEliece, D.V. Sarwate, On Sharing Secrets and Reed–Solomon Codes. *Communications of the ACM*, 9(24):583–584, (1981)

[29] S. Micali, P. Rogaway, Secure computation. Unpublished manuscript, 1992. Preliminary version, in *CRYPTO'91*. LNCS 576 (Springer, 1991), pp. 392–404

[30] M. Pease, R. Shostak, L. Lamport, Reaching Agreement in the Presence of Faults. In the *Journal of the ACM*, 27(2):228–234, (1980)

[31] T. Rabin, M. Ben-Or, Verifiable secret sharing and multi-party protocols with honest majority, in *21st STOC* (1989), pp. 73–85

[32] A. Shamir, How to Share a Secret. In the *Communications of the ACM*, 22(11):612–613, (1979)

[33] A.C. Yao, Theory and application of trapdoor functions, in *23rd FOCS* (1982), pp. 80–91

[34] A. Yao, How to generate and exchange secrets, in *27th FOCS* (1986), pp. 162–167