CHAPTER 4

# Asynchronous secure computation

We study secure multiparty computation in asynchronous networks. (See an introductory presentation in Section 1.4.) We first present, in Section 4.1, our definition of asynchronous secure multiparty computation. We consider the bounded variant of the secure channels setting, in the presence of adaptive adversaries and non-erasing parties. Here we also describe our conventions for presenting asynchronous protocols. Next we present, in Section 4.2, asynchronous primitives used in our protocols. In Sections 4.3 and 4.5 we describe our construction of the cases of Fail-Stop and Byzantine adversaries, respectively. In Section 4.4 we define and construct an Asynchronous Verifiable Secret Sharing (AVSS) scheme, that is a key tool in our construction for Byzantine adversaries. In Section 4.6 we demonstrate the optimality of our constructions.

## 4.1 Preliminaries

### 4.1.1 The asynchronous model

Consider an asynchronous network of $n$ parties, where every two parties are connected via a reliable and private communication channel. Messages sent on a channel can be arbitrarily delayed; however, each message sent is eventually received. Furthermore, the order in which messages are received on a channel may be different from the order in which they were sent[1].

It is convenient to regard a computation in our model as a sequence of **steps**. In each step a single party is active. The party is activated by receiving a message; it then performs an internal computation, and possibly sends messages on its outgoing channels. We consider the order of the steps as controlled by an adversarial entity, called a **scheduler**. We allow computationally unbounded schedulers. The privacy of the channels is modelled by considering only **oblivious** schedulers. The only information known to these schedulers is the origin and destination (and, possibly, the length) of each message sent[2]. More formally,

[1] For simplicity, we assume that messages are not duplicated. Duplication can be prevented by standard methods, e.g., using counters

[2] Private channels are useless in the presence of schedulers that have access to the *contents* of the messages:

an **oblivious scheduler** is a function$D : (\mathbf{N} \times [n]^2)^* \to \mathbf{N}$, with the following interpretation. Given the list of $\{$*length, source, destination*$\}$ of the $i$ first messages sent in an execution, in some standard global ordering, function $D$ specifies the serial number of the next message to be delivered. (The scheduler must deliver every message exactly once, and cannot deliver unsent messages.) In the sequel we incorporate the scheduler within the adversary.

We distinguish two types of adversaries: **Fail-Stop** and **Byzantine**. If the adversary is Fail-stop, then the corrupted parties may stop sending messages at some time during the computation; however, we assume that the corrupted parties continue to receive messages and have an output. If the adversary is Byzantine, then the corrupted parties may deviate from their protocols in any way. We consider adaptive adversaries.

### 4.1.2 Defining secure asynchronous computation

We use the 'ideal model methodology', described in Section 1.2 and Chapter 2, for defining asynchronous secure computation. The 'asynchronous version' of the ideal model should reflect the special properties of the asynchronous setting. In particular, the following property of the asynchronous setting remains unchanged, even in the presence of a trusted party. In an asynchronous network with $t$ potential corruptions, the uncorrupted parties (as well as the trusted party) cannot wait to communicate with more than $n-t$ parties before deciding on the output of a computation, since up to $t$ parties may never join the computation. Consequently, unlike synchronous computations, the output of an asynchronous computation can be based only on some 'core' subset, of size at least $n-t$, of the inputs. Furthermore, the $t$ inputs that were left out are not necessarily inputs of corrupted parties: they can be inputs of slow, however uncorrupted parties.[3] [4]

Therefore, we suggest the following asynchronous ideal model. Evaluating a function $f$ in this model proceeds in the same way as in the synchronous ideal model (see Section 2.3 on page 21), with the only exception that the computation stage is modified as follows.

**Computation stage (asynchronous version):** The adversary chooses an arbitrary 'core' set of size at least $n-t$; this subset, denoted $C$, is independent of the inputs of the uncorrupted parties. A 'scheduler' delivers only the messages of the parties in $C$ to the trusted party.

Upon receiving the (possibly substituted) inputs of the parties in subset $C$, the trusted party computes some predefined "approximation" to the function value, based on $C$ and the inputs of the parties in $C$. (For concreteness, we use the following "approximation": set the inputs of the parties in $\bar{C}$ to 0, and compute the original function.) In order for the output to make more sense, the trusted party outputs the subset $C$ as well as the approximated function value.

---

for instance, such a scheduler can set the delivery order of the messages so that the first bit in the first message received by party $P_1$ will be the same as the fifth bit in the first message that party $P_2$ sent on its private channel to $P_3$... thus 'breaking' the privacy of this channel.

[3] Chor and Moscovici describe this property of the asynchronous model in more detail [CM]. Furthermore, they give an exact characterization of the achievable 'tasks' in the presence of a given number of Fail-Stop corruptions, when the privacy of the inputs is disregarded.

[4] A consequence of this phenomenon is that no asynchronous computation can be "equivalent" to a computation in the *synchronous* trusted party scenario.

Consequently, the output of the parties in this ideal model is as follows. On input $\vec{x} = x_1, \ldots, x_n$ of the parties (input $x_i$ to party $P_i$), let $C$ be the set described above, and let $\vec{y}$ denote the (possibly modified) inputs of the parties in $C$. Let $f_C$ denote the "approximation" of the function $f$, made by the trusted party. Then, the uncorrupted parties output $(C, f_C(\vec{y}))$. The corrupted parties output an arbitrary function of the information gathered by the adversary in the computation; this information consists of the identity and inputs of the corrupted parties, their random inputs, and the resulting approximation $f_C(\vec{y})$. Also here, we let the $n$-vector $\text{IDEAL}_{f,\mathcal{S}}(\vec{x}) = \text{IDEAL}_{f,\mathcal{S}}(\vec{x})_1 \ldots \text{IDEAL}_{f,\mathcal{S}}(\vec{x})_n$ denote the outputs of the parties on input $\vec{x}$ and ideal-model adversary $\mathcal{S}$ (party $P_i$ outputs $\text{IDEAL}_{f,\mathcal{S}}(\vec{x})_i$).

In Definitions 4.1 through 4.3 we present notations for the output of the parties in the asynchronous ideal model. (These notations capture the above description in an analogous way to Definitions 2.8 through 2.10 in the synchronous setting, and can be skipped in a first reading.) We use the same technical notations as in Section 2.3:

- For a vector $\vec{x} = x_1 \ldots x_n$ and a set $C \subseteq [n]$, let $\vec{x}_C$ denote the vector $\vec{x}$, projected on the indices in $C$.

- For an $n$-vector $\vec{x} = x_1 \ldots x_n$, a set $B \subseteq [n]$, and a $|B|$-vector $\vec{b} = b_1 \ldots b_{|B|}$, let $\vec{x}/_{(B,\vec{b})}$ denote the vector constructed from vector $\vec{x}$ by substituting the entries in $B$ by the corresponding entries from $\vec{b}$.

Using these notations, the *approximation of function $f$ based on a subset $C \subseteq [n]$* is defined by $f_C(\vec{x}) = f(\vec{x}/_{(\bar{C},\vec{0})})$.

**Definition 4.1** *Let $D$ be the domain of possible inputs of the parties, and let $\mathcal{R}$ be the domain of possible random inputs. A $t$-*limited asynchronous ideal-model-adversary *is a quintuple $\mathcal{S} = (t, b, h, c, O)$, where:*

- *$t$ is the maximum number of corrupted parties.*

- *$b : [n]^* \times D^* \times \mathcal{R} \to [n] \cup \{\perp\}$ is the* selection *function for corrupting parties (the value $\perp$ is interpreted as "no more parties to corrupt at this stage")*

- *$h : [n]^* \times D^* \times \mathcal{R} \to D^*$ is the* input substitution *function*

- *$c : A^* \times \mathcal{R} \to \{C \subseteq [n] \mid |C| \geq n - t\}$ is a* core set selection *function,*

- *$O : D^* \times \mathcal{R} \to \{0,1\}^*$ is an* output *function for the bad parties.*

The sets of corrupted parties are now defined as follows.

**Definition 4.2** *Let $D$ be the domain of possible inputs of the parties, and let $\mathcal{S} = (t, b, h, c, O)$ be a $t$-limited ideal-model-adversary. Let $\vec{x} \in D^n$ be an input vector, and let $r \in \mathcal{R}$ be a random input for $\mathcal{S}$. The $i$*th set of corrupted parties *in the ideal model, denoted $B^{(i)}(\vec{x}, r)$, is defined as follows.*

- *$B^{(0)}(\vec{x}, r) = \phi$*

- *Let $b_i \stackrel{\triangle}{=} b(B^{(i)}(\vec{x}, r), \vec{x}_{B^{(i)}(\vec{x},r)}, r)$. For $0 \leq i < t$, and as long as $b_i \neq \perp$, let*

$$B^{(i+1)}(\vec{x}, r) \stackrel{\triangle}{=} B^{(i)}(\vec{x}, r) \cup \{b_i\}$$

- *Let $i^*$ be the minimum between $t$ and the first $i$ such that $b_i = \perp$. Let $b_i^f \triangleq b(B^{(i)}(\vec{x}, r), \vec{x}_{B^{(i)}(\vec{x},r)}, f_C(\vec{y}), r)$, where $\vec{y}$ is the substituted input vector for the trusted party, and $C$ is the selected core subset. That is, $\vec{y} \triangleq \vec{x}/_{(B^{(i^*)}(\vec{x},r), h(B^{(i^*)}(\vec{x},r), \vec{x}_{B^{(i^*)}(\vec{x},r)}, r))}$, and $C \triangleq c(\vec{x}_{B^{(i^*)}(\vec{x},r)}, r)$.*

  *For $i^* \le i < t$, let*
$$B^{(i+1)}(\vec{x}, r) \triangleq B^{(i)}(\vec{x}, r) \cup b_i^f.$$

In Definition 4.3 we use $B^{(i)}$ instead of $B^{(i)}(\vec{x}, r)$.

**Definition 4.3** *Let $f : D^n \to D'$ for some domains $D$ and $D'$ be the computed function, and let $\vec{x} \in D^n$ be an input vector. The* output of computing function $f$ in the asynchronous ideal model *with adversary $\mathcal{S} = (t, b, h, c, O)$, on input $\vec{x}$ and random input $r$, is an $n$-vector $\text{IDEAL}_{f,\mathcal{S}}(\vec{x}) = \text{IDEAL}_{f,\mathcal{S}}(\vec{x})_1 \ldots \text{IDEAL}_{f,\mathcal{S}}(\vec{x})_n$ of random variables, satisfying for every $1 \le i \le n$:*
$$\text{IDEAL}_{f,\mathcal{S}}(\vec{x})_i = \begin{cases} (C, f_C(\vec{y})) & \text{if} \quad i \notin B^{(t)} \\ O(\vec{x}_{B^{(t)}}, f_C(\vec{y}), r) & \text{if} \quad i \in B^{(t)} \end{cases}$$
*where $r$ is the random input of $\mathcal{S}$, $B^{(t)}$ is the $t^{\text{th}}$ set of corrupted parties, $\vec{y} = \vec{x}/_{(B^{(t)}, h(B^{(t)}, \vec{x}_{B^{(t)}}, r))}$ is the substituted input vector for the trusted party, and $C \triangleq c(\vec{x}_{(B^{(t)}}, r)$ is the selected core subset.*

Next we describe the execution of a protocol $\pi$ in the asynchronous real-life scenario, where the uncorrupted parties run some semi-honest protocol $\pi'$ for $\pi$, and in the presence of a computationally unbounded, adaptive $t$-limited asynchronous real-life adversary $\mathcal{A}$. (We incorporate the scheduler, described in Section 4.1.1, in $\mathcal{A}$.) The computation proceeds as follows. Initially, and upon the receipt of a message, each party sends messages according to $\pi'$. The adversary (playing the scheduler) sees the sender, receiver and length of each message sent. It also sees the contents of the messages sent to corrupted parties. After the sending of each message, the adversary may decide to deliver one of the messages that were sent and not yet delivered. It may also decide to corrupt some parties. Upon corruption of a party $P$, the adversary sees all the data kept by $P$ according to $\pi'$. As in Section 2.3, we assume that the corrupted parties output the adversary view of the comutation (i.e., all the information gathered by the adversary during the computation). We let $\text{VIEW}_{\pi,\mathcal{A}}(\vec{x})$ and $\text{EXEC}_{\pi,\mathcal{A}}(\vec{x})$ have an analogous meaning to that of Section 2.3, with respect to the asynchronous setting.

In Definition 4.4 below we concentrate on the bounded variant of the (asynchronous) secure channels setting. (See Section 2.1 for a presentation of the variants.)

**Definition 4.4** *Let $f : D^n \to D'$ for some domains $D$ and $D'$, and let $\pi$ be a protocol for $n$ parties that runs in* PPT. *We say that $\pi$* asynchronously $t$-securely computes $f$ in the bounded secure channels setting, *if the following requirements hold for any semi-honest protocol $\pi'$ for $\pi$ (according to one of the Definitions 2.5 through 2.7), and for any asynchronous $t$-limited real-life adversary $\mathcal{A}$:*

**Termination:** *On all inputs, all the uncorrupted parties complete execution of the protocol with probability 1.*

**Security:** *there exists an asynchronous $t$-limited ideal-model-adversary $\mathcal{S}$, whose complexity is polynomial in the complexity of $\mathcal{A}$, such that for every input vector $\vec{x}$,*

$$\text{IDEAL}_{f,\mathcal{S}}(\vec{x}) \stackrel{\text{d}}{=} \text{EXEC}_{\pi',\mathcal{A}}(\vec{x}).$$

**Remarks:**

- We remark that the Termination property is implicit in the security property (parties that did not complete the protocols do not have an output). We explicitly require Termination to stress the importance and delicacy of this requirement in an asynchronous setting. In particular, we note that Consensus is a (very limited) special case of secure computation. The [FLP] impossibility result for *deterministic* protocols implies that in an asynchronous network with potential faults there must exist non-terminating runs of any (randomized) protocol reaching Consensus. Thus, we only require a secure protocol to terminate with probability (or measure) 1.
- If only Fail-Stop adversaries are allowed, then the real-life adversary $\mathcal{A}$ only specifies: (a) the conditions upon which the corrupted parties should stop sending messages, and (b) the output of the corrupted parties. Consequently, the input substitution function $h$ of the trusted-party adversary are the identity function.

### 4.1.3 Writing asynchronous protocols

We review the way in which an asynchronous computation is carried out, and describe some useful writing conventions for asynchronous protocols. In Section 4.1.1 we described an asynchronous computation as a sequence of **steps**. This is a global, 'external' view of the computation. From the point of view of a party, a computation is a sequence of **cycles**. A cycle is initiated upon receiving a message; it consists of executing an internal computation, and possibly sending messages to other parties. Once a cycle is completed, the party waits for the next message. The set of instructions to be executed upon the receipt of a message is called an **asynchronous protocol**. These instructions may depend on the internal state of the party, and on the contents of the message received.

However, for clarity of presentation and analysis, our description of an asynchronous protocol is somewhat different: we partition the protocol into several modules, called **sub-protocols**. Each sub-protocol is first presented, and sometimes analyzed, as if it were the only protocol run by the party. Sub-protocols are combined by letting one sub-protocols 'call' other sub-protocols during its execution. We present our interpretation of this writing style of asynchronous protocols.

The party keeps track of the sub-protocols that are currently 'in execution': initially, only one predefined sub-protocol is 'in execution'; when some sub-protocol is called by another sub-protocol, it is added to the sub-protocols 'in execution'.

We assume that each message applies to one sub-protocol only. Thus, Upon the receipt of a message, the party invokes a cycle of the relevant sub-protocol. If the received message refers to a sub-protocol which is not 'in execution', then the party keeps the message; once the sub-protocol has started, a cycle is invoked for each one of the kept messages.

We associate input and output values with each sub-protocol; furthermore, the output of one sub-protocol may be the input of another sub-protocol. However, the latter sub-protocol may be in execution before the former sub-protocol has completed updating its output; consequently, the input of the latter sub-protocol may be changed during its execution. We call such input a dynamic input; namely, unlike regular input, dynamic input variables may have different input values in different cycles in the same run of a sub-protocol.

The variables we use for dynamic input will only have items added to them. It is convenient to regard such variables as monotonically increasing sets. Namely, let $\mathcal{U}$ be such a variable, and let $\mathcal{U}^{(c)}$ denote the set held in $\mathcal{U}$ when this set is of size $c$ for the first time; then, if $c < d$ we have $\mathcal{U}^{(c)} \subseteq \mathcal{U}^{(d)}$. We call these variables accumulative sets. (In the sequel, we will use calligraphic letters (e.g., $\mathcal{U}$) to denote accumulative sets.)

Using the above conventions, the shorthand "Set $a = \pi(b)$" stands for: "(a) call sub-protocol $\pi$, with $b$ for input; (b) Let $a$ denote its output". Note that $b$ may be a dynamic input of $\pi$; similarly, the output $a$ may be updated as long as sub-protocol $\pi$ is in execution.

Another shorthand used is as follows. Let *condition* denote some relation (e.g. 'there are at least $k$ elements in the accumulative set'). We use "Wait until *condition*" to denote "If *condition* holds, continue to the next instruction. Else, end this cycle." In the sequel we do not distinguish between protocols and sub-protocols.

## 4.2 Primitives

In this section we describe primitives used in our constructions, in both the Fail-Stop and the Byzantine cases. We first define the requirements of each primitive, and then describe (or give a reference to) an implementation.

### 4.2.1 Byzantine Agreement

We present the standard definition of asynchronous Byzantine Agreement. (When the adversary is Fail-Stop, this primitive is sometimes called Consensus.)

**Definition 4.5** *Let $\pi$ be an $n$-party protocol where each party has binary input. Protocol $\pi$ is a $t$-*resilient Byzantine Agreement *protocol if the following hold, for every input, every scheduler, and every coalition of up to $t$ corrupted parties.*

- **Termination.** *With probability 1, all the uncorrupted parties eventually complete the protocol (i.e., terminate locally).*
- **Correctness.** *All the uncorrupted parties that complete the protocol have an identical output. Furthermore, if all the uncorrupted parties have the same input, denoted $\sigma$, then all the uncorrupted parties output $\sigma$.*

Feldman [Fe] describes an $(\lceil \frac{n}{4} \rceil - 1)$-resilient asynchronous BA protocol, running in constant expected time[5]. $(\lceil \frac{n}{3} \rceil - 1)$-resilient Consensus can be reached by substituting the AVSS scheme in Chapter 5 by a simple secret sharing scheme. (For instance, use the scheme described in Section 4.3.)

[5] We define the running time of an asynchronous protocol in Section 4-A.

### 4.2.2 Broadcast

We present a definition of a broadcast protocol.

**Definition 4.6** *Let $\pi$ be an $n$-party protocol initialized by a special party (called the sender), having input $m$ (the message to be broadcast). Protocol $\pi$ is a $t$-*resilient Broadcast *protocol if the following hold, for every input, scheduler, and coalition of up to $t$ corrupted parties.*

- **Termination.** 1. *If the sender is uncorrupted, then all the uncorrupted parties eventually complete the protocol.*
  2. *If any uncorrupted party completes the protocol, then all the uncorrupted parties eventually complete the protocol.*
- **Correctness.** *If the uncorrupted parties complete the protocol, then they do so with a common output $m^*$. Furthermore, if the sender is uncorrupted then $m^* = m$.*

We stress that the Termination property of Broadcast is much weaker than the Termination property of Byzantine Agreement: for Broadcast, we do not require that the uncorrupted parties complete the protocol, in case that the sender is corrupted.

In Figure 4-1 we describe a simple Broadcast protocol for the Fail-Stop model.

**Protocol BC**

Code for the sender (on input $m$):

1. Send $(\text{MSG}, m)$ to all the parties, and complete the protocol with output $m$.

Code for the other parties:

2. Upon receiving the *first* $(\text{MSG}, m)$ or $(\text{ECHO}, m)$ message, send $(\text{ECHO}, m)$ to all the parties and complete the protocol with output $m$.

Figure 4-1: A simple broadcast protocol for Fail-stop adversaries

**Proposition 4.7** *Protocol BC is an $n$-resilient Broadcast protocol for Fail-stop adversaries.*

**Proof:** If the sender is uncorrupted, then all uncorrupted parties receive a $(\text{MSG}, m)$ message and thus complete the protocol with output $m$. If a uncorrupted party completed the protocol, with output $m$, then it has sent an $(\text{ECHO}, m)$ message to all the parties, thus every uncorrupted party will complete the protocol with output $m$. □

Bracha [Br] describes an $(\lceil \frac{n}{3} \rceil - 1)$-resilient Broadcast protocol for the Byzantine setting. For self-containment, we present Bracha's Byzantine Broadcast (BB) protocol in Figure 4-2.

*Convention:* in the sequel, we use "party $P$ received an $m$ broadcast" to shorthand "party $P$ completed a Broadcast protocol with output $m$". We assume that the identity of the sender appears in $m$.

**Protocol BB**

<u>Code for the sender (on input $m$):</u>

1. send message $(\text{MSG}, m)$ to all the parties.

<u>Code for party $P_i$:</u>

2. Upon receiving a message $(\text{MSG}, m)$, send $(\text{ECHO}, m)$ to all the parties.
3. Upon receiving $n-t$ messages $(\text{ECHO}, m')$ that agree on the value of $m'$, send $(\text{READY}, m')$ to all the parties.
4. Upon receiving $t+1$ messages $(\text{READY}, m')$ that agree on the value of $m'$, send $(\text{READY}, m')$ to all the parties.
5. Upon receiving $n-t$ messages $(\text{READY}, m')$ that agree on the value of $m'$, send $(\text{OK}, m')$ to all the parties and accept message $m'$ as a broadcast message.

Figure 4-2: Bracha's Broadcast protocol

### 4.2.3 Agreement on a Core Set

We first illustrate a setting in which an Agreement on a Core Set (ACS) primitive is needed. Assume each party initiates a Broadcast of some value; next, the parties wish to agree on a *common* 'core' set of at least $n-t$ parties whose Broadcast has been successfully completed. Clearly, each uncorrupted party can wait to (locally) complete $n-t$ Broadcasts, and keep the senders of these Broadcasts. However, if more than $n-t$ Broadcasts have been (globally) completed, then we cannot be sure that all the uncorrupted parties keep the same set of parties. For this purpose, the parties use the Agreement on a Core Set (ACS) primitive: the parties' output of this primitive, when used in this context, is a common set of at least $n-t$ parties whose Broadcast has been completed.

We proceed to formally define the properties required of an ACS primitive. Recall the definition of an accumulative set (described in Section 4.1.3, on page 53): we let $\mathcal{U}^{(c)}$ denote the value of variable $\mathcal{U}$ in cycle $c$ of the protocol; An accumulative set $\mathcal{U}$ is a variable holding a set, such that for every two cycles $c$ and $d$, if $c$ precedes $d$ then $\mathcal{U}^{(c)} \subseteq \mathcal{U}^{(d)}$.

**Definition 4.8** *Let $m, M \in \mathbf{N}$ (in out context, we use $m = n-t$ and $M = n$), and let $\mathcal{U}_1 \ldots \mathcal{U}_n \subseteq [M]$ be a collection of accumulative sets, so that party $P_i$ has $\mathcal{U}_i$. We say that the collection is $(m,t)$-*uniform*, if the following hold, for every scheduler and every coalition of up to $t$ corrupted parties.*

- *For every uncorrupted party $P_i$ there exists a cycle $c$ such that $|\mathcal{U}_i^{(c)}| \geq m$.*
- *For every two uncorrupted parties $P_i$ and $P_j$, if $k \in \mathcal{U}_i^{(a)}$ for some cycle $a$ within party $P_i$, then there exists a cycle $b$ within party $P_j$ so that $k \in \mathcal{U}_j^{(b)}$ (namely, $P_i$ and $P_j$ will eventually have $\mathcal{U}_i = \mathcal{U}_j$).*

**Definition 4.9** *Let $m \leq M$, and let $\pi$ be an $n$-party protocol with parameters $m, M$, where the input of every uncorrupted party $P_i$ is an accumulative set $\mathcal{U}_i$. Protocol $\pi$ is a $t$-*resilient protocol for Agreement on a Core Set *(with parameters $m, M$), if the following hold, for every coalition of up to $t$ corrupted parties, and every scheduler.*

- **Termination.** *If the collection $\mathcal{U}_1 \ldots \mathcal{U}_n$ is $(m,t)$-uniform, then with probability 1 all the uncorrupted parties eventually complete the protocol.*
- **Correctness.** *All the uncorrupted parties complete the protocol with a common output $C \subseteq [M]$ so that $|C| \geq m$. Furthermore, every uncorrupted party has $C \subseteq \mathcal{U}_i^*$, where $\mathcal{U}_i^*$ is the value of $\mathcal{U}_i$ upon the completion of protocol $\pi$.*

Before presenting our construction, let us remark that in [BKR] a simpler construction of an $(\lceil \frac{n}{3} \rceil - 1)$-reslient protocol for agreement on a core set is described. Furthermore, the [BKR] protocol runs in constant expected time. Let $n \geq 3t+1$. Our construction, with parameters $m$ and $M$, consists of two phases:

**Phase I:** In the first phase, each party first waits until its dynamic input is of size $m$; then, it performs $\log_2 n$ iterations. In each iteration, the party sends the current contents of its dynamic input to all the other parties; then, the party collects the sets sent by the other parties in this iteration, and waits until its dynamic input contains $n-t$ such sets; then, the party continues to the next iteration. It will be shown below that the intersection of the dynamic inputs of all the uncorrupted parties that have completed this phase is of size at least $m$. [6]

**Phase II:** In the second phase, the parties concurrently run $M$ Byzantine Agreement protocols; in the $c$th Byzantine Agreement, the parties decide whether element $c \in [M]$ will be in the agreed set $C$. Namely, each party's input to the $c$th Byzantine Agreement is 1 iff $c$ belongs to the party's dynamic input; the element $c$ belongs to the set $C$, iff the (common) output of the $c$th Byzantine Agreement is 1. The properties of Byzantine Agreement assure us that the set $C$ contains the intersection of the dynamic inputs of all the parties that have completed the first step; therefore, the set $C$ is large enough.

**Remark:** Our construction applies to both the Fail-Stop and the Byzantine cases, using the appropriate agreement primitive (either BA or Consensus).

Protocol ACS is described in Figure 4-3.

**Proposition 4.10** *Protocol $ACS[m, M]$ is a $min(r, \lceil \frac{n}{3} \rceil - 1)$-resilient protocol for Agreement on a Core Set in a network of $n$ parties, where $r$ is the resilience of the agreement protocol used.*

**Proof:** We first assert the Termination condition. Assume that the accumulative sets $\mathcal{U}_1 \ldots \mathcal{U}_n$ define an $(m,t)$-uniform collection. We show that every uncorrupted party $P_i$ will eventually complete the protocol.

Clearly, Step 1 will be completed. It can be seen, by induction on the number of iterations, that every iteration in Step 2 will be completed: in each iteration $r$, party $P_i$ will eventually receive the $(r, S_j)$ message from every uncorrupted party. Furthermore, for every

[6]This problem of assuring a large intersection of the sets of a uniform collection was previously addressed by [Fe], as well as [BE]. Both works present partial solutions to this problem: let $m = n - t$; Feldman [Fe] made sure that the intersection of the sets held by all the uncorrupted parties is of size $\Theta(n)$ (but not necessarily $m$); Ben-Or and El-Yaniv [BE] made sure that the intersection of the sets held by $n - 2t$ uncorrupted parties is of size at least $m$.

**Protocol ACS**$[m, M](\mathcal{U}_i)$

Party $P_i$ acts as follows, on inputs $m, M$ and accumulative set $\mathcal{U}_i$.

1. Wait until $|\mathcal{U}_i| \geq m$.
2. For $0 \leq r \leq \lfloor \log n \rfloor$ do
   (a) Send $(r, \mathcal{U}_i)$ to all the parties.
   (b) Let $F_i^r = \{P_j |$ an $(r, S_j)$ message was received from $P_j\}$.
   Wait until $S_j \subseteq \mathcal{U}_i$ for at least $n - t$ parties $P_j \in F_i^r$.
   (Note that sets $S_j$ that were not contained in $\mathcal{U}_i$ when they were received, may later be contained in $\mathcal{U}_i$ if elements are added to it.)
3. Run $M$ Byzantine Agreement protocols $BA_1 \ldots BA_M$, with input 1 to $BA_j$ iff $j \in \mathcal{U}_i$.
4. Set $C_i = \{j |$the output of $BA_j$ is $1\}$. Wait until $C_i \subseteq \mathcal{U}_i$.
5. Output $C_i$.

Figure 4-3: ACS - A protocol for Agreement on a Core Set

uncorrupted party $P_j$, party $P_i$ will eventually have $S_j \subseteq \mathcal{U}_i$ (since the collection $\mathcal{U}_1 \ldots \mathcal{U}_n$ is uniform).

Step 3 will be completed with probability 1, since all the Byzantine Agreement protocols are completed with probability 1. To see that Step 4 will be completed, we note that if a Byzantine Agreement protocol $BA_j$ has output 1, then there exists a uncorrupted party $P_k$ that started $BA_j$ with $j \in \mathcal{U}_k$; thus, party $P_i$ will eventually have $j \in \mathcal{U}_i$.

We assert the Correctness property. The unanimity of the outputs of the parties follows directly from the definition of Byzantine Agreement. Step 4 of the protocol assures us that $C \subseteq \mathcal{U}_i^*$, within each uncorrupted party $P_i$. It is left to show that $|C| \geq m$.

Let $U_i^r$ denote the value of accumulative set $\mathcal{U}_i$ upon completion of iteration $r$ of Step 2 by party $P_i$. We show by induction on $r$ that every set $D$ of at most $2^r$ uncorrupted parties, all of which have completed iteration $r$, satisfies $|\cap_{j \in D} U_j^r| \geq m$.

The base of induction ($r = 0$) is immediate from Step 1 of the protocol. For the induction step, consider a set $D$ of up to $2^r$ uncorrupted parties. We first observe that for every two parties $P_j, P_k \in D$, we have $|F_j^r \cap F_k^r| \geq t + 1$, since $n \geq 3t + 1$ and $|F_j^r| \geq n - t$ for each $j$. Therefore, there exists at least one *uncorrupted* party $P_l \in F_j^r \cap F_k^r$; we call $P_l$ an *arbiter* of $P_j$ and $P_k$. Party $P_j$ (resp. $P_k$) has received the $(r, S_l)$ message; furthermore, $U_l^{r-1} \subseteq S_l$. Thus, $U_l^{r-1} \subseteq U_j^r$ (resp. $U_l^{r-1} \subseteq U_k^r$).

Consider an arbitrary partition of the set $D$ to pairs of parties, choose an arbiter for each pair, and let $D'$ be the set of these arbiters; thus, $|D'| \leq 2^{r-1}$. By the induction hypothesis, we have $m \leq |\cap_{j \in D'} U_j^{r-1}|$. Every party in $D$ has an arbiter in $D'$, thus $\cap_{j \in D'} U_j^{r-1} \subseteq \cap_{j \in D} U_j^r$. Therefore, we have $m \leq |\cap_{j \in D} U_j^r|$.

Let $D$ be the set of uncorrupted parties that start Step 3 of the protocol, and let $\hat{C} = \cap_{j \in D} U_j^{\log n}$; by the above induction, we have $|\hat{C}| \geq m$. For every $j \in \hat{C}$, the inputs of all the uncorrupted parties to the BA$_j$ protocol is 1. Therefore, by the definition of Byzantine Agreement, the output of every BA$_j$ protocol is 1. Thus, $\hat{C} \subseteq C$ and $|C| \geq m$.

□

## 4.3 Fail-Stop faults

We show how to securely $t$-compute any function whose input is partitioned among $n$ parties, when $n \geq 3t+1$ and the faults are Fail-Stop. Our costruction follows the outline of the first [BGW] construction (namely, the construction resilient against corrupted parties which only try to gather information but otherwise follow the protocol).

Let $F$ be a finite field known to the parties with $|F| > n$, and let $f : F^n \rightarrow F$ be the computed function. (Namely, we restrict our discussion to functions where the input of each party, as well as the function value, are field elements. We note that no generality is lost in this restriction: if the range of the computed function does not 'fit into' $F$, then the function can be partitioned into several functions whose range is $F$. Furthermore, a party can have more than one field element for input; in this case, it executes a different copy of the protocol for each field element.)

We assume that the parties have an arithmetic circuit computing $f$; the circuit consists of addition and multiplication gates of in-degree 2 (we regard adding a constant as a special case of addition). All the computations in the sequel are done in $F$.

An outline of the protocol follows. Let $x_i$ be the input of $P_i$. As a first step, each party shares its input among the parties, using a technique similar to Shamir's secret sharing scheme [Sh]. (Namely, for each party $P_i$ that successfully shared its input, a random polynomial $p_i(\cdot)$ of degree $t$ is generated, so that each party $P_j$ has $p_i(j)$ and $p_i(0)$ is $P_i$'s input value. We say that $p_i(j)$ is $P_j$'s share of $p_i(0)$.) Next, the parties agree, using protocol ACS, on a core set $C$ of parties that have successfully shared their input. Once $C$ is computed, the parties proceed to compute $f_C(\vec{x})$, in the following way. First, the input values of the parties not in $C$ are set to a default value, say 0; then, the parties evaluate the given circuit, gate by gate, in a way described below. Note that the output of the protocol is fixed once the set $C$ is fixed.

For each gate, the parties use their shares of the input lines to jointly and securely 'generate' a random polynomial $p(\cdot)$ of degree $t$ such that every party $P_i$ computes $p(i)$, and $p(0)$ is the output value of this gate. (Namely, $p(i)$ is $P_i$'s share of the output line of this gate.) Once enough parties have computed their shares of the output line of the entire circuit, the parties reveal their shares of the output line, and interpolate the output value.

In the rest of this section, we describe our construction in more detail. First, we describe the Global-Share and Reconstruct protocols. Next, we describe the evaluation of a linear gate and of a multiplication gate. Finally, we put all the pieces together to describe the main protocol, and prove its correctness.

When presenting each protocol, we describe the uncorrupted parties' outputs of the protocol. We postpone the formal proof of security to Section 4.3.5.

### 4.3.1 Global-Share and Reconstruct

The Global-Share protocol, denoted GShare, consists of two phases. First, each party shares a secret among the parties; next, the parties use an ACS protocol in order to agree on a set, of size at least $n-t$, of parties that have successfully shared their secret. Party $P_i$'s output of the GShare protocol is the set of parties that have successfully shared their inputs, along with the $i$th share of the input of each of the parties in this set.

This protocol is applied once at the beginning of the protocol (where each party shares its input of the main protocol), and twice in each invocation of the protocol for computing a multiplication gate (see Section 4.3.3). Protocol GShare has a 'security parameter', $d$. This parameter will be set to either $t$ or $2t$, according to the context in which the protocol is activated. Protocol GShare is presented in Figure 4-4.

**Protocol GShare**$[d](x_i)$

Party $P_i$ acts as follows, on inputs $x_i, d$.

1. "Share $x_i$":
   (a) Choose a random polynomial $h_i(\cdot)$ of degree $d$, such that $h_i(0) = x_i$.
   For each $1 \leq j \leq n$, send $h_i(j)$ to party $P_j$.
   (b) Broadcast `'Party` $P_i$ `completed sharing'`.
2. Upon receiving the share $s_{i,j}$ of $P_j$'s secret and the `'Party` $P_j$ `completed sharing'` broadcast, add $j$ to a set $C_i$ of parties that have successfully shared their secret.
   Set $C$ =ACS$[n-t, n](C_i)$.
3. Output $C$, and $\{s_{i,j} | j \in C\}$.

Figure 4-4: GShare - The global sharing protocol

We note that the accumulative sets $C_1 \dots C_n$ of Step 2 define an $(n-t, t)$-uniform collection. Thus, all the uncorrupted parties complete protocol ACS (and, hence, the entire GShare protocol) with the desired output. Furthermore, the corrupted parties have no information about the values shared by the uncorrupted parties.

In the Reconstruct protocol, described in Figure 4-5, the parties reconstruct a secret from its shares. The parameters of this protocol are the 'security parameter' $d$, and a set $R$ of the parties to which the secret is to be revealed. Party $P_i$'s input is the $i$-th share of the secret, denoted $s_i$. The Reconstruct protocol is invoked once for every multiplication gate, and once at the end of the protocol. (Parameter $R$ is necessary in the multiplication gate invocation. In the other invocation it is set to $R = [n]$.)

We note that if $d + 1 \leq n - t$ then a Reconstruct protocol that is run by all the parties will be completed. Furthermore, if there exists a polynomial, $p(\cdot)$, of degree $d$ such that the share of each active party $P_j$ is $s_j = p(j)$, then all the uncorrupted parties in $R$ will output $p(0)$.

### 4.3.2 Evaluating a linear gate

We describe the evaluation of a linear gate, instead of a simple addition gate; this more general formulation will be convenient in presenting the protocol for computing a multiplication gate.

Evaluating a linear gate is easy and requires no communication. Let $c = \sum_{j=1}^{k} \alpha_j \cdot a_j$ be a linear gate, where $a_1 \dots a_k$ are the input lines of the gate, $\alpha_1 \dots \alpha_k$ are fixed coefficients,

**Protocol Reconstruct**$[d, R](s_i)$

Party $P_i$ acts as follows, on input $s_i$, and parameters $d \in \{t, 2t\}$ and $R \subseteq \{P_1, \ldots, P_n\}$.

1. Send $s_i$ to all the parties in $R$.
2. If $P_i \notin R$, output 0.
   Otherwise, upon receiving $d+1$ values (value $v_j$ from party $P_j$), interpolate a polynomial $p_i(\cdot)$ of degree $d$ such that $p(j) = v_j$ for each received value $v_j$.
   Output $p_i(0)$.

Figure 4-5: Reconstruct - The reconstruction protocol

and $c$ is the output line. Let $A_j(\cdot)$ be the polynomial associated with the $j$th input line; namely, party $P_i$'s share of this line is $a_{i,j} = A_j(i)$.

Each party $P_i$ locally sets its share of the output line to $c_i = \sum_{j=1}^{k} \alpha_j \cdot a_{i,j}$. It can be easily seen that the shares $c_1 \ldots c_n$ define a random polynomial $C(\cdot)$ of degree $t$ with the correct free coefficient, and with $C(i) = c_i$ for all $i$.

### 4.3.3 Evaluating a multiplication gate

Let $c = a \cdot b$ be a multiplication gate, and let $A(\cdot), B(\cdot)$ be the polynomials associated with the input lines; namely, each party $P_i$'s shares of these lines are $A(i)$ and $B(i)$, respectively. The parties will jointly compute their shares of a random polynomial $C(\cdot)$ of degree $t$, satisfying $C(0) = A(0) \cdot B(0)$; namely, each uncorrupted party $P_i$'s share of the output line will be $C(i)$.

Initially, each party locally computes its share of the polynomial $E(\cdot) = A(\cdot) \cdot B(\cdot)$, by setting $E(i) = A(i) \cdot B(i)$. Clearly, $E(\cdot)$ has the required free coefficient. However, $E(\cdot)$ is of degree $2t$; moreover, it is not uniformly distributed. The parties use their shares of $E(\cdot)$ to compute their shares of the desired polynomial $C(\cdot)$, in a secure manner. The computation proceeds in two steps: first, the parties jointly generate a *random* polynomial, $D(\cdot)$, of degree $2t$, so that $D(0) = E(0)$ (namely, each party $P_i$ will have $D(i)$). Next, the parties will use their shares of $D(\cdot)$ in order to jointly compute their shares of polynomial $C(\cdot)$. These steps are described below.

**Randomization.** We describe how polynomial $D(\cdot)$ is generated. First, the parties generate a random polynomial $H(\cdot)$ of degree $2t$ and with $H(0) = 0$; namely, each party $P_i$ will have $H(i)$.

We first describe how the polynomial $H(\cdot)$ is shared in a synchronous setting [BGW]. Each party $P_i$ selects a random polynomial $H_i(\cdot)$ of degree $2t$ with $H_i(0) = 0$, and shares it among the parties; namely, each party $P_j$ receives $H_i(j)$. Polynomial $H(\cdot)$ is set to $H(\cdot) = \sum_{j=1}^{n} H_j(\cdot)$; namely, each party $P_i$ computes $H(i) = \sum_{j=1}^{n} H_j(i)$.

In our asynchronous setting, a party cannot wait to receive its share from all the other parties. Instead, the parties agree, using protocol ACS, on a set $C$ of parties that have successfully shared their $H_i$ polynomials; next, polynomial $H(\cdot)$ will be set to $H(\cdot) =$

$\sum_{j\in C} H_j(\cdot)$. In other words, the parties run protocol GShare[2t](0); on output $(C, \{H_j(i)|j \in C\})$ of the GShare protocol, party $P_i$ computes $H(i) = \sum_{j\in C} H_j(i)$.

Polynomial $D(\cdot)$ is now defined as $D(\cdot) = E(\cdot) + H(\cdot)$; namely each party $P_i$ computes $D(i) = E(i) + H(i)$. Clearly, $D(0) = A(0) \cdot B(0)$, and all the other coefficients of $D(\cdot)$ are uniformly and independently distributed over $F$.

**Degree-reduction.** In the degree-reduction step, the parties use their shares of $D(\cdot)$ in order to jointly and securely compute their shares of a random polynomial $C(\cdot)$ of degree $t$, with $C(0) = D(0)$. Polynomial $C(\cdot)$ will be set to the 'truncation' of polynomial $D(\cdot)$ to degree $t$; namely, the $t+1$ coefficients of $C(\cdot)$ are the coefficients of the $t+1$ lower degrees of $D(\cdot)$. An important observation is that the information gathered by the corrupted parties (namely, $t$ shares of polynomial $D(\cdot)$, along with $t$ shares of the truncation polynomial $C(\cdot)$), is independent of $C(0)$. This statement is formalized in Lemma 4.16 on page 70, and used in proving the security of the entire protocol.

Let $\vec{d} = D(1)\ldots D(n)$, and let $\vec{c} = C(1)\ldots C(n)$. [BGW] noted that there exists a fixed $n \times n$ matrix $M$ such that $\vec{c} = \vec{d}M$. It follows that the desired output of each party $P_i$ is a linear combination of the inputs of the parties: $C(i) = [\vec{d}M]_i = \sum_{j=1}^{n} D(i) \cdot M_{j,i}$. We colloquially call such an operation 'multiplying the inputs by a fixed matrix'. (In this case, the parties' inputs are their shares of the polynomial $D(\cdot)$.)

In a synchronous setting, 'multiplying the inputs by a fixed matrix' can be done by securely computing the appropriate $n$ fixed linear combinations of the inputs, so that the value of the $i$th linear combination is revealed to party $P_i$ only. Linear combinations are securely computed as follows. First, each party shares its input; next, each party computes the linear combination of its shares (as in Section 4.3.2), and reveals this linear combination to the specified party; finally, the specified party computes the output value by interpolating a degree $t$ polynomial from the received combinations. [7]

Linear combinations of *all* the inputs cannot be computed in an asynchronous setting. Thus, the synchronous method described above cannot be used. We outline our solution for the asynchronous setting. First, we describe a technique for multiplying inputs by a fixed matrix in an asynchronous setting, for the case where the matrix and the set of inputs are related in a special way described below. Next, we note that the matrix $M$ and the set of possible inputs of the degree-reduction step are related in this special way.

**Definition 4.11** *Let $A$ be an $n \times n$ matrix, and let $S \subseteq F^n$ be a set of input vectors. we say that $S$ is $t$-*multipliable* by $A$, if for every set $G \subseteq [n]$ with $|G| \geq n - t$, there exists an (easily computable) matrix $A_G$, of size $|G| \times n$, such that for every input $\vec{x} \in S$, we have $\vec{x}_G \cdot A_G = \vec{x} \cdot A$.*

Let $S$ be $t$-multipliable by $A$. Then, the protocol described in Figure 4-6 'multiplies inputs in $S$ by the matrix $A$'. Let $\vec{x} \in S$. The parties first execute a GShare protocol (applied on the input vector $\vec{x}$); once the common set $G$ is computed, each party locally computes $A_G$. Next, the parties run $n$ Reconstruct protocols; in the $i$th Reconstruct, the

[7] Note that computing a linear combination of input values is harder than computing a linear combination of already-shared values (i.e., the problem addressed in section 4.3.2): the former computation involves *sharing* of all the inputs.

parties let $P_i$ compute $[\vec{x}_G \cdot A_G]_i$, by sending him the appropriate linear combination of their shares.

**Protocol MAT**$(x_i, A)$

Party $P_i$ acts as follows, on input $x_i$, and matrix $A$.

1. Set $(G, \{s_{i,j} | j \in G\})$ =GShare$[t](x_i)$.
   Enumerate $G = g_1, \ldots, g_{|G|}$, and let $\vec{s_i} = s_{i,g_1} \ldots s_{i,g_{|G|}}$.
2. Compute $A_G$.
3. For $1 \leq k \leq n$, set $y_k$ =Reconstruct$([\vec{s_i} \cdot A_G]_k, t, \{k\})$.
   Output $y_i$.

Figure 4-6: MAT - A protocol for multiplying the input by a fixed matrix

We say that an input vector $\vec{x}$ is $d$-**generated** if there exists a polynomial $P(\cdot)$ of degree $d$ satisfying $x_i = P(i)$ for every $i$; the set of possible inputs of the degree-reduction step is the set of $2t$-generated vectors.

**Proposition 4.12** *Let $M$ be the $n \times n$ matrix introduced in* [BGW]. *Then, the set of $2t$-generated $n$-vectors is $t$-multipliable by $M$, when $n \geq 3t + 1$.*

**Proof:** Recall that matrix $M$ used in [BGW] is constructed as $M = V^{-1}TV$, where $V$ is a (Vandermonde) $n \times n$ matrix defined by $V_{i,j} = i^j$, and $T$ is constructed by setting all but the first $t + 1$ rows of a Unit matrix to 0. (Let $\vec{d}, \vec{c}$ be as defined above. To see that $\vec{d} \cdot M = \vec{c}$, note that $\vec{d} \cdot V^{-1}$ is the coefficients vector of the polynomial $D(\cdot)$; thus $\vec{d} \cdot V^{-1}T$ is the coefficients vector of $C(\cdot)$, and $\vec{d} \cdot V^{-1}TV = \vec{c}$.)

Let $G \subseteq [n]$ with $|G| \geq n - t$, and let $G = g_1 \ldots g_{|G|}$. Matrix $M_G$ is constructed as follows. Let the $|G| \times |G|$ matrix $V^G$ be the matrix $V$ projected on the indices in $G$; namely, $V^G_{i,j} = (g_i)^j$. Next, construct the $|G| \times n$ matrix $\hat{V}$ by appending $n - |G|$ zero columns to $(V^G)^{-1}$. Finally, set $M_G = \hat{V}TV$, where $T$ is defined above.

Since $n \geq 3t + 1$, we have $|G| \geq 2t + 1$; it can be verified that in this case, $\vec{x}_G \cdot \hat{V}$ is, once again, the coefficients vector of $D(\cdot)$ (namely, $\vec{x}_G \cdot \hat{V} = \vec{x} \cdot V^{-1}$). Thus, $\vec{x}_G \hat{V} TV = \vec{x} \cdot V^{-1}TV = \vec{x} \cdot M$ as required. □

Combining the Randomization and Degree-reduction steps, we derive a protocol for computing a multiplication gate. This protocol, denoted MUL, is presented in Figure 4-7.

### 4.3.4 The main protocol

Let $f : F^n \to F$ be given by an arithmetic circuit $A$. Our protocol for securely $t$-computing $f$ is described in Figure 4-8.

**Theorem 4.13** *Let $f : F^n \to F$, for some field $F$ with $|F| > n$, and let $A$ be a circuit computing $f$. Then, protocol* FScompute$[A]$ *asynchronously* $(\lceil \frac{n}{3} \rceil - 1)$*-securely computes $f$ in the bounded secure channels setting with $n$ non-erasing parties, provided that only Fail-Stop adversaries are considered.*

**Protocol MUL**$(a_i, b_i)$

Party $P_i$ acts as follows, on inputs $a_i, b_i$.

1. Set $(C', \{h_{i,j} | j \in C'\})$ =GShare$[2t](0)$.
   Let $d_i = a_i \cdot b_i + \sum_{j \in C'} h_{i,j}$.
2. Let $M_{n \times n}$ be the matrix introduced in [BGW].
   Set $c_i$ =MAT$(d_i, M)$.
   Output $c_i$.

Figure 4-7: MUL - A protocol for computing a multiplication gate

**Protocol FScompute**$[A](x_i)$

Party $P_i$ acts as follows, on local input $x_i$, and given circuit $A$.

1. Set $(C, \{s_{i,j} | j \in C\})$ =GShare$[t](x_i)$.
   For a line $l$ in the circuit, let $l^{(i)}$ denote the share of party $P_i$ in the value of this line. If $l$ is the $j$th input line of the circuit, then set $l^{(i)} = s_{i,j}$ if $j \in C$, and $l^{(i)} = 0$ otherwise.
2. For each gate $g$ in the circuit, wait until the $i$th shares of all the input lines of $g$ are computed. Then, do the following.
   (a) If $g$ is an addition gate with output line $l$ and input lines $l_1$ and $l_2$, then set $l^{(i)} = l_1^{(i)} + l_2^{(i)}$.
   (b) If $g$ is a multiplication gate $l = l_1 \cdot l_2$, then set $l^{(i)}$ =MUL$(l_1^{(i)}, l_2^{(i)})$.
3. Let $l_{out}$ be the output line of the circuit.
   Once $l_{out}^{(i)}$ is computed, send 'Ready' to all the parties.
   (The only role of the 'Ready' messages is simplifying the proof of correctness, below.)
4. Wait to receive $n - t$ 'Ready' messages.
   Set $y$ =Reconstruct$[t, [n]](l_{out}^{(i)})$.
5. Output $(C, y)$.

Figure 4-8: FScompute - The protocol for Fail-Stop faults

We present the proof of Theorem 4.13 in two steps. First, we show security of the protocol in the presence of non-adaptive adversaries. (When the adversary is non-adaptive there is no difference between the cases where the uncorrupted parties are semi-honest or honest. We thus assume that the uncorrupted parties are honest.) We do not present a definition of non-adaptively, secure computation in asynchronous networks. However, such a definition can be easily assembled from the definition of adaptive security in asynchronous networks (Definition 4.4), and definition of non-adaptive security in synchronous networks (Definition 2.4).

Next we show adaptive security of protocol FScompute in the presence of non-erasing parties. We remark that protocol FScompute can be shown adaptively secure also in the presence of honest-looking parties, using similar constructs to those described in Section 3.4.

### 4.3.5 Proof of correctness — non-adaptive case

let $\mathcal{A}$ be a non-adaptive Fail-stop adversary, and let $B$ be the set of corrupted parties (we incorporate the scheduler in $\mathcal{A}$).

#### Termination

We use FS to shorthand protocol FScompute$[A]$. We show that for every input $\vec{x}$, with probability 1 all the uncorrupted parties terminate protocol FS.

Protocol FS consists of several invocations of protocols GShare and Reconstruct; if a uncorrupted party terminates all these invocations, then this party terminates the entire protocol as well. We have seen that a Reconstruct protocol always terminates, provided that $n \geq 3t+1$. If all the Consensus protocols invoked in a GShare protocol terminate, then the GShare protocol terminates as well.

Each one of these Consensus protocols terminates with probability 1, by the definition of Consensus. Thus, with probability 1, all the Consensus protocols invoked in the entire protocol terminate; therefore, with probability 1, all the GShare protocols terminate as well.

#### Security

For any real-life non-adaptive $(\lceil \frac{n}{3} \rceil - 1)$-limited real-life adversary $\mathcal{A}$, we construct a ideal-model adversary $\mathcal{S}$, so that for every input $\vec{x}$, we have $\text{IDEAL}_{f,\mathcal{S}}(\vec{x}) = \text{EXEC}_{\text{FS},\mathcal{A}}(\vec{x})$. (The notations $\mathcal{S}$ and $\text{IDEAL}_{f,\mathcal{S}}(\cdot)$, as well as $\text{EXEC}_{\pi,\mathcal{A}}(\cdot)$ for a protocol $\pi$ (here $\pi$ =FS), are defined in Section 4.1.2, on page 23.) Here the set of corrupted parties, denoted $B$, is fixed. We represent the ideal-model adversary $\mathcal{S}$ as a quadruple $A = (B, h, c, O)$.

**Construction of the ideal-model adversary.** We remark that adversary $\mathcal{S}$ operates via black-box simulation of the real-life adversary. (Black-box simulation was described, for the synchronous setting, in Section 2.3. The asynchronous version is analogous.) Here the set $B$ of parties corrupted by the real-life adversary $\mathcal{A}$ is fixed. Adversary $\mathcal{S}$ corrupts the parties in $B$. The input substitution function $h$, the core set selection function $c$ and the output function $O$ are computed via the following simulated interaction between the real-life adversary $\mathcal{A}$ and parties executing protocol FS.

On input $\vec{x}_B$ and random input $r$, $\mathcal{S}$ extends $\vec{x}_B$ to an $n$-vector $\vec{x}'$ by filling the extra entries with, say, zeros. Partition $r$ into $n$ sections $r_1 \ldots r_n$, and let $\vec{r} = r_1 \ldots r_n$. Simulate an execution of protocol FS with adversary $\mathcal{A}$, and with $x'_i$ as input and $r_i$ as random input of each party $P_i$. (In the sequel we incorporate $D$ in $\mathcal{A}$). Let $C$ be the core set decided upon in the GShare invocation in Step 1 of the protocol. For each $P_i \in C$, let $y_i$ be the value shared by $P_i$ (namely, $y_i = x_i$); for $P_i \notin C$, let $y_i = 0$. Let $\vec{y} = y_1 \dots y_n$. Set $h(\vec{x}_B, r) = \vec{y}_B$, and $c(\vec{x}_B, r) = C$. [8]

Our next step is computing the output function $O$ of the corrupted parties in the ideal model. The inputs to this function are $\vec{x}_B$, $r$, and $f_C(\vec{y})$. (Recall that $f_C(\vec{y})$ is the value received from the trusted party). We continue the above simulation of protocol FS, with the following modification. Step 3 of the protocol assures us that by the time the first uncorrupted party starts executing the Reconstruct protocol of Step 4, at least $t+1$ uncorrupted parties have computed their shares of the output line. In the proof of Lemma 4.15 below, we show that these shares determine a unique polynomial of degree $t$; let $p(\cdot)$ denote this polynomial.[9] Once the first uncorrupted party reaches the final Reconstruct protocol, the simulator computes the above polynomial $p(\cdot)$, and chooses a random polynomial, $p'(\cdot)$, of degree $t$, such that $p'(0) = f_C(\vec{y})$, and $p'(i) = p(i)$ for every *corrupted* party $P_i$. In the simulated run, every *uncorrupted* party $P_j$ executes the last Reconstruct protocol with input $p'(j)$, instead of $p(j)$. Let $w$ denote the output of the corrupted parties after this simulated run of protocol FS. Set $O(\vec{x}_B, r, f_C(\vec{y})) = w$. [10]

**Validity of the ideal-model adversary.** It is left to show that for every input vector $\vec{x}$, the output vector $\text{EXEC}_{\text{FS},\mathcal{A}}(\vec{x})$ of the parties in the 'real' computation has the same distribution as the output vector $\text{IDEAL}_{f,\mathcal{S}}(\vec{x})$ of the parties in the ideal model. An outline of our proof follows. Recall the definition of the **adversary view** of the computation (we re-define this notion in more detail below). This notion captures the information gathered by the adversary during the computation, combined with the information seen by the scheduler. In Lemma 4.14 below, we show that the adversary view of the real computation is distributed equally to the adversary view in the interaction simulated by the ideal-model adversary. We complete our proof by showing that whenever the adversary view of the real execution equals the adversary view of the simulated execution, the output vector of *all* the parties in the real-life model equals the output vector of *all* the parties in the ideal model.

We proceed to re-define the adversary view. First, define the **transcript** of a computation. The transcript $U$ consists of the inputs $\vec{x}$ of all the parties, their random inputs $\vec{r}$, and the entire communication among the parties. The communication is organized in **events**; each event, $e$, consists of a message received by a party, and its corresponding responses. The order of events is induced by the order of delivery of the messages. Namely, we have

$$U = \vec{x}, \vec{r}, e_1, \dots, e_k.$$

We partition each message in protocol FS to its **contents** and its **frame**. The contents of

---

[8] Since the faults are Fail-Stop, we could let the input substitution function $h$ be the identity function. However, this formulation is valid for the proof of Theorem 4.30 (Security of the Byzantine protocol) as well. Note that $f(\vec{y}) = f_C(\vec{y})$; in the sequel, we leave the subscript $C$ for clarity.

[9] Step 3 of the algorithm is a technical step whose purpose is making the fact that $p(\cdot)$ is fixed at this stage more obvious. It can be seen that, even without Step 3, by the time the first uncorrupted party starts executing the last Reconstruct protocol, at least $n - 2t$ uncorrupted parties have computed their shares of the output line.

[10] The output of the corrupted parties is defined as the contents of their output tapes after all the messages sent by the uncorrupted parties have been received.

a message consists of the elements of the field $F$, appearing in this message. The frame of a message is the message itself where each field element is replaced by a 'blank'. In particular, the frame contains the type of the message, the name of the protocol and invocation that the message belongs to, and the identity of the sender and the receiver. Messages that do not contain any field elements have empty contents. (Informally, the scheduler sees only the frame of each message sent, while the receiver sees the entire message.) [11]

Let $S$ be a set of parties, and let $U = \vec{x}, \vec{r}, e_1, \ldots, e_k$ be a transcript of some computation. The **view** of the computation, as seen by the parties in $S$, is the sequence $U_S = \vec{x}_S, \vec{r}_S, \hat{e}_1, \ldots, \hat{e}_k$, where $\vec{x}_S$ is the inputs of the parties in $S$ and $\vec{r}_S$ is their random inputs. Each $\hat{e}_i$ is a **semi-event**; namely, if the recipient party of event $e_i$ is in $S$, then $\hat{e}_i$ consists of the full received message and the subsequent responses. Otherwise, $\hat{e}_i$ consists of only the frame of the received message and responses. The **adversary view** is the view of the set $B$ of corrupted parties. We note that the adversary view contains all the information gathered by the adversary during the relevant computation. Without loss of generality, we assume that the corrupted parties' output of a computation is the adversary view. Let $\hat{O}$ denote this function.

Fix an input vector $\vec{x}$. Define the random variable $\mu$: for each random input $\vec{r}$, set $\mu$ to be the corrupted parties' view of the execution of protocol FS on input $\vec{x}$, random input $\vec{r}$, and adversary $\mathcal{A}$. Let the random variable $\mu'$ be similarly defined, with respect to the simulated protocol described above. In Lemma 4.14 below we show that $\mu$ and $\mu'$ are identically distributed.

**Lemma 4.14** *The random variables $\mu$ and $\mu'$ are identically distributed.*

(The proof of Lemma 4.14 is given at the end of this section.)

It is left to show that the following two quantities are equal, with respect to every possible view, $V$, of the corrupted parties (i.e., every view $V$ in the support set of $\mu$ and $\mu'$):[12]

**(a)** The output of *all* the parties in the ideal model with the adversary $\mathcal{S}$ described above, when the corrupted parties' view of the simulated execution is $V$.

**(b)** The output of *all* the parties after an execution of protocol FS, when the view of the corrupted parties is $V$.

Equality is proven by showing that in both cases, the output of the corrupted parties is $\hat{O}(V)$, and the output of the uncorrupted parties is $(C, f_C(\vec{y}))$, where $C$ is the core set appearing in $V$ and $\vec{y}$ is the substituted input vector defined above. (Note that both $C$ and $\vec{y}$ are uniquely determined by $\vec{x}$ and $V$).

Case **(a)** (the ideal model): By the definition of the output of the parties in the ideal model (Definition 2.10 on page 23), the uncorrupted parties output $(C, f_C(\vec{y}))$, and the corrupted parties output $O(\vec{x}_B, r, f_C(\vec{y}))$; in the above construction of $\mathcal{S}$, we have set $O(\vec{x}_B, r, f_C(\vec{y})) = \hat{O}(V)$.

[11] Note that this partitioning of a message to its contents and frame is specific to protocol FS. A more general definition may let the frame of a message be its length only. However, the present definition of a frame is more natural for our protocol; furthermore, it simplifies our proof of Lemma 4.14 below.

[12] We use $V$ to shorthand $U_B$.

Case **(b)** (an execution of protocol FS): The corrupted parties output $\hat{O}(V)$, by the definition of function $\hat{O}$. In Lemma 4.15 below, we show that the uncorrupted parties output $f_C(\vec{y})$. $\square$

**Lemma 4.15** *Consider an execution of protocol FS on input $\vec{x}$ with adversary $B$, and let $C$ and $\vec{y}$ be the quantities defined above, with respect to this execution. Then, the uncorrupted parties' output of this execution is $f_C(\vec{y})$.*

**Proof:** For each line $l$ in the computed circuit, let $v_l$ be the value of this line when the input of the circuit is $\vec{y}$. Let $l_{out}$ denote the output line of the computed circuit; since the circuit computes the function $f$, we have $v_{l_{out}} = f(\vec{y}) = f_C(\vec{y})$.

Consider an execution of protocol FS as in the Lemma. For each line, $l$, let $p_l(\cdot)$ be the lowest degree polynomial such that the share of each uncorrupted party $P_i$ in this line, in this execution, is $p_l(i)$. It can be seen, by induction on the structure of the circuit, that every $p_l(\cdot)$ is of degree at most $t$ and that $p_l(0) = v_l$: each line is either an input line, an output of a linear gate, or an output of a multiplication gate. These cases were dealt with in the sections describing the initial commit, the linear gate, and the multiplication gate protocol, respectively. Thus, in the final Reconstruct protocol, the uncorrupted parties retrieve (and output) $p_{l_{out}}(0) = v_{l_{out}} = f_C(\vec{y})$. $\square$

**Proof of Lemma 4.14.** We show, by induction on the number of communication events, that every prefix of $\mu$ has the same distribution as the corresponding prefix of $\mu'$ with the same number of events.

We use the following notations. For every $i \geq 0$, let $\mu_i$ denote the prefix of $\mu$ that ends after the $i$th semi-event. Let $\mu'_i$ be similarly defined with respect to $\mu'$. Fix an instance, $V_i$, of $\mu_i$, and let $\epsilon_{i+1}$ be the random variable describing the distribution of the $(i+1)$th semi-event in $\mu$, given that the corresponding prefix of $\mu$ is $V_i$. Namely, for each semi-event $\hat{e}$,

$$\text{Prob}(\epsilon_{i+1} = \hat{e}) = \text{Prob}(\mu_{i+1} = (V_i, \hat{e}) \mid \mu_i = V_i).$$

Recall that each semi-event consists of a received message and the subsequent response messages. Let $\sigma_{i+1}$ and $\tau_{i+1}$ be the random variables having the distributions of the received message and the response messages in $\epsilon_{i+1}$, respectively. Let $\epsilon'_{i+1}$, $\sigma'_{i+1}$, and $\tau'_{i+1}$ be similarly defined with respect to $\mu'$.

For the base of induction, we note that $\vec{x}_B = \vec{x}'_B$, and both random inputs $\vec{r}_B$ and $\vec{r}'_B$ are uniformly distributed; thus, $(\vec{x}_B, \vec{r}_B)$ and $(\vec{x}'_B, \vec{r}\,')$ are identically distributed.

For the induction step, we show that for every $i \geq 0$ and for every instance $V_i$, the random variables $\epsilon_{i+1}$ and $\epsilon'_{i+1}$ are identically distributed.

First, note that $V_i$ contains all the information seen by the scheduler when delivering the $(i+1)$st message. Therefore, the frame of the $(i+1)$st message is uniquely determined by $V_i$. In particular, the sender (resp. the recipient) of the $(i+1)$st message is the same party in both computations. We distinguish two cases:

**(a)** The recipient party is uncorrupted. In this case, it is left to show that the frames of the response messages sent in this event are identically distributed. However, it can be seen, by observing the protocol, that the frames of the messages sent by a uncorrupted party are uniquely determined by the frames of the messages received by this party so far; these frames are part of the common instance $V_i$ of the prefixes $\mu_i$ and $\mu'_i$.

**(b)** The recipient party is corrupted. Clearly, if the received messages $\sigma_{i+1}$ and $\sigma'_{i+1}$ are identically distributed, then the response messages $\tau_{i+1}$ and $\tau'_{i+1}$ are identically distributed as well. If the sender is corrupted, then entire received message is explicitly written in the instance $V_i$, which is the same in both computations. It remains to consider the case in which the sender is uncorrupted. We have that the *frames* of $\sigma_i$ and $\sigma'_i$ are equal. We show that the *contents* of $\sigma_i$ and $\sigma'_i$ are identically distributed.

Every message in protocol FS belongs either to some invocation of protocol GShare, or to some invocation of protocol Reconstruct. We consider these two cases separately. In the sequel, we say that a random variable is a **semi-random polynomial** of degree $d$, if its instances are polynomials of degree at most $d$, such that all the coefficients other than the free coefficient are uniformly and independently distributed. (The free coefficient may have any distribution.)

**GShare messages.** The only messages of a GShare protocol having non-empty contents are shares of some secret. However, the corrupted parties receive only up to $t$ shares of each secret shared by a uncorrupted party, namely up to $t$ shares of a semi-random polynomial of degree $t$. Consequently, both in $\mu$ and in $\mu'$ the contents is distributed uniformly and independently over the field $F$, regardless of the shared values.

In an invocation of Reconstruct, the corrupted parties receive $n$ shares of a semi-random polynomial $p(\cdot)$ of degree $t$. Thus, the contents of the first $t$ messages are uniformly distributed, both in $\mu$ and in $\mu'$. The contents of the other messages are uniquely determined by the $t$ shares known to the corrupted parties (namely, the contents of the first $t$ messages of the same invocation), and the free coefficient of $p(\cdot)$. Thus, it suffices to show that the free coefficient of $p(\cdot)$ (namely, the output of this invocation of Reconstruct) is identically distributed in $\mu$ and $\mu'$. We distinguish two types of invocations of Reconstruct: a Reconstruct within a multiplication step (namely, a Reconstruct invoked in Step 3 of protocol MAT), and the final invocation of Reconstruct, in Step 4 of protocol FS.

**Reconstruct as a part of a multiplication step.** Consider an invocation of protocol MUL (see Figure 4-7 on page 64). We show that the corrupted parties' outputs of the $t$ invocations of Reconstruct, along with the contents of the other messages of this invocation of MUL, received by the corrupted parties, are uniformly and independently distributed, regardless of the values of the input lines of the corresponding multiplication gate.

The polynomial $H(\cdot)$ generated by the parties in the Randomization phase (Step 1 of protocol MUL) is a semi-random polynomial of degree $2t$, with zero free coefficient. The contents of the GShare messages of Step 1 received by the corrupted parties add up to $t$ shares of the polynomial $H(\cdot)$. Fix some arbitrary polynomials $A(\cdot)$ and $B(\cdot)$ associated with the input lines of the corresponding multiplication gate, and let $D(\cdot) = H(\cdot) + A(\cdot) \cdot B(\cdot)$. Then, both in $\mu$ and in $\mu'$, $D(\cdot)$ is a semi-random polynomial of degree $2t$ with some fixed free coefficient.

The data gathered by the corrupted parties during the entire invocation of MUL consists of $t$ shares of the semi-random polynomial $D(\cdot)$, along with $t$ shares of the truncation polynomial $C(\cdot)$ (namely, the outputs of the $t$ invocations of Reconstruct). Let $s = A(0) \cdot B(0)$. Lemma 4.16 below shows that there exists a *unique* instance of $D(\cdot)$ (namely, a unique polynomial of degree $2t$ whose free coefficient is $s$), that corresponds to each sequence of $2t$ field elements gathered by the corrupted parties. Consequently, the sequence of $2t$ field-elements gathered by the corrupted parties is uniformly distributed, both in $\mu$ and in $\mu'$. [13]

**Final Reconstruct.** Let $C$ be the core set determined by $V_i$, and let $\vec{y}$ be the substituted input vector described above. By Lemma 4.15, the parties' output of the final Reconstruct in a computation of protocol FS is $f_C(\vec{y})$. By the construction of the ideal-model adversary $\mathcal{S}$, the parties' output of the final Reconstruct in the simulated execution is also $f_C(\vec{y})$.

This concludes our proof that $\mu$ and $\mu'$ have the same distribution. □

**Lemma 4.16** (technical lemma). *Let $F$ be a finite field with $|F| > d$, and let $s \in F$. Then for every sequence $v_1, \ldots, v_d, u_1, \ldots, u_d$ of field-elements, there exists a unique polynomial $p(\cdot)$ of degree $2d$ with $p(0) = s$, such that:*

1. *For $1 \leq i \leq d$, we have $p(i) = v_i$.*
2. *For $1 \leq i \leq d$, we have $q(i) = u_i$, where $q(\cdot)$ is the truncation of $p(\cdot)$ to degree $d$. (Namely, the coefficients of $q(\cdot)$ are the coefficients of the $d+1$ lower degrees of $p(\cdot)$.)*

**Proof:** See Section 4-B. □

### 4.3.6 Proof of correctness — adaptive case

Let $\mathcal{A}$ be a $t$-limited adaptive Fail-stop adversary, where $n \geq 3t+1$ (we incorporate the scheduler within the adversary). Termination is shown as in the non-adaptive case, provided that the Consensus protocol in use is resilient against adaptive adversaries. To show Security, we construct an ideal-model adversary $\mathcal{S} = (t, b, h, c, O)$, as in Definition 4.1. We remark that our construction of $\mathcal{S}$ uses only with black-box access to $\mathcal{A}$. Here we assume non-erasing parties. That is, each party keeps all the information gathered during the computation. This information consists of the party's input and random input, and the messages received from other parties.

**Construction of the ideal-model adversary.** The functions $b, h, c, O$ are determined via a simulated execution of protocol FS with adversary $\mathcal{A}$. That is, $\mathcal{S}$ hands $\mathcal{A}$ information on the computation, in an interactive process that simulates a real-life computation. (We use the description of a real-life computation presented immediately above Definition 4.4 on page 52.)

Recall the definition of the contents and frame of a message (presented on page 66). The information handed to $\mathcal{A}$ in the simulated interaction is as follows.

1. $\mathcal{S}$ hands $\mathcal{A}$ the *frame* of each message sent by each party according to protocol FS. (This is the information seen by the scheduler. We note that protocol FS has the following property. At any point during the computation, the frames of the messages

[13] In Lemma 4.16 we assume, without loss of generality, that the corrupted parties are $P_1, \ldots, P_t$.

received by a party so far determine the frame of the next message sent by this party. Thus, the frames of the messages sent by the uncorrupted parties are predictable by the adversary.)

2. Up to the **decision point**, defined below, The *contents* of each message sent from an uncorrupted party to a corrupted party is set to uniformly chosen random elements in the field $F$.

3. Up to the decision point, whenever the adversary decides to corrupt a party $P$, the simulator $\mathcal{S}$ proceeds as follows.

   First, $\mathcal{S}$ hands $\mathcal{A}$ simulated contents of all the messages previously received by $P_i$. The simulated contents of the messages sent by corrupted parties, including parties that were uncorrupted at the time of sending but became corrupted since, are consistent with the contents already known to the adversary. The simulated contents of messages sent by still uncorrupted parties are chosen uniformly from $F$.

   Next $\mathcal{S}$ decides to corrupt $P_i$ in the ideal-model (in the first corruption stage). Upon learning $x_i$, the input of $P_i$, the simulator chooses, in a way described below, a **simulated random input** $r_i$ for $P_i$ that is consistent with input $x_i$, with the contents of the messages previously received by $P_i$ (as chosen above), and with the messages previously sent by $P$ to corrupted parties. Note that $r_i$ determines also the messages that were supposedly sent by $P_i$ to *un*corrupted parties. Finally $\mathcal{S}$ hands $\mathcal{A}$ the input $x_i$ and the simulated random input $r_i$.

Functions $b, c, h$ are determined as follows. Let the **decision point** be the time where the first uncorrupted party invokes the **final Reconstruct** (i.e., the Reconstruct of Step 4 of protocol FS, see Figure 4-8). The selection function $b()$ of corrupted parties is determined as described in Step 3 above. Let $B$ be the set of corrupted parties at the decision point. Let $C$ be the core set decided upon in the initial GShare (i.e., in the GShare of Step 1 of the protocol). For each $P_i \in C$, let $y_i$ be the value shared by $P_i$ (namely, $y_i = x_i$); for $P_i \notin C$, let $y_i = 0$. Let $\vec{y} = y_1 \ldots y_n$. Set $h(\vec{x}_B, r) = \vec{y}_B$, and $c(\vec{x}_B, r) = C$.

The output function $O$ of $\mathcal{S}$ is determined as follows. At the decidion point $\mathcal{S}$ asks the trusted party for the computed function value. $\mathcal{S}$ continues the above simulation of protocol FS, with the following modifications. At the decision point, $\mathcal{S}$ computes each corrupted party's share of the output line of the circuit. (This can be efficiently done based on $\mathcal{S}$'s information on the computation.) Next, $\mathcal{S}$ chooses a random polynomial $p'(\cdot)$ such that $p'(0) = f_C(\vec{y})$ and for each corrupted party $P_j$, the value $p'(j)$ equals $P_j$'s share of the output line. (The polynomial $p'(\cdot)$ plays a similar role as in the non-adaptive case.) In the simulated run, each uncorrupted party $P_i$ executes the final Reconstruct protocol with input $p'(i)$. If the adversary decides to corrupt $P_i$ after the decision point, then $\mathcal{S}$ proceeds as in Step 3 above, with the exception that the chosen random input $r_i$ should be consistent also with $P_i$ having $p'(i)$ as input to the final Reconstruct. (In the ideal model these corruptions are part of the second corruption stage.) Let $w$ denote the output of the corrupted parties after this simulated run of protocol FS, and let $B'$ be the set of faulty parties in this run. Set $O(\vec{x}_{B'}, r, f_C(\vec{y})) = w$.

It remains to describe how the random input $r_i$ is chosen, upon the corruption of a party $P_i$. Assume first that $P_i$ is corrupted before the decision point. We distinguish the following cases.

**Random input for the initial invocation of GShare:** Here the relevant information known to the adversary consists of up to $t-1$ uniformly chosen elements in $F$, sent by $P_i$ to corrupted parties. $\mathcal{S}$ chooses a random polynomial $q()$ of degree $t$, such that $q(0) = x_i$ (where $x_i$ is $P_i$'s input), and for each corrupted party $P_j$, the value of $q()$ at point $j$ equals the value sent by $P_i$ to $P_j$. We note that $q()$ can be chosen efficiently. Furthermore, $q()$ has the distribution of a semi-random polynomial of degree $t$ with $q(0) = x_i$. (This is so since the values sent by the simulated $P_i$ to corrupted parties were uniformly chosen from $F$.) $\mathcal{S}$ sets the random input of $P_i$ for the initial invocation of GShare so that the polynomial chosen in Step 1(a) of GShare (See Figure 4-4) is $q()$.

We remark that the messages received by $P_i$ from other parties during this invocation of GShare are irrelevant to $P_i$'s random choices.

**Random input for an invocation of MULT:** The relevant information gathered by $\mathcal{A}$ on $P_i$ in an invocation of MULT is as follows.

- $P_i$'s shares of the input lines of the corresponding gate. These shares are determined by the simulated invocations of the procedures for evaluating the input lines of this gate.
- Up to $t-1$ shares of the degree-$2t$ polynomial chosen by $P_i$ in the invocation of GShare in the Randomization Step (Step 1 of MULT, see Figure 4-7). These are the shares sent by $P_i$ to corrupted parties.
- $P_i$'s shares of the polynomials shared by other parties in this invocation of GShare. ($\mathcal{S}$ hands $\mathcal{A}$ these uniformly chosen shares upon the corruption of $P_i$, as described in Step 2 above.)
- Up to $t-1$ shares of the degree-$t$ polynomial chosen by $P_i$ in the Degree-Reduction Step (Step 1 of MAT, see Figure 4-6). These are the shares sent by $P_i$ to corrupted parties.

Let the **fingerprint** of an invocation of MULT within $P_i$ consist of this information seen by $\mathcal{A}$. Each fingerprint is consistent with an equal number of random inputs of this invocation of MULT (i.e., with an equal number of polynomials chosen by $P_i$ in Steps 1 of MULT and MAT). This can be seen in a similar way to Lemma 4.16: Each fingerprint can be completed, in the same number of ways, to a sequence that uniquely determines the polynomials chosen by $P_i$. Furthermore, as remarked in the proof of Lemma 4.16, sampling these polynomials with the appropriate probability, given a fingerprint, can be done efficiently.

If $P_i$ is corrupted after the decision point, then it may be the case that $P_i$ has already started the final Reconstruct, and has sent $p'(i)$ to corrupted parties. In this case, $P_i$'s simulated random input should be consistent with $P_i$ having $p'(i)$ for its share of the output line of the circuit. Such random input can be efficiently sampled, in a similar way to the previous case (i.e., the case where $P_i$ is corrupted before the decision point.)

We note that randomness is used also in the various invocations of Consensus. $\mathcal{S}$ uses, and hands $\mathcal{A}$ upon corruption of $P_i$, uniformly chosen random inputs for the use of $P_i$ in the simulated invocations of Consensus.

**Validity of the ideal-model adversary.** The validity of the simulation is shown in a similar way to the non-adaptive case, with the exception that an additional type of event is added to the adversary view of the computation: corruption of some party. Consequently, the case of corruption of a party has to be considered in the inductive proof of Lemma 4.14. In Lemma 4.17 we state the inductive claim of Lemma 4.14, for the case of a corruption event. A full proof of this lemma is omitted.

**Lemma 4.17** *Consider $n$ parties running protocol* FS *on some input $\vec{x}$. Let $\mu_i$ consist of the first $i$ events in some instance $\mu$ of the adversary view of this computation, such that the $(i+1)$th event in $\mu$ is the corruption of party $P_l$. Let $c^{(l)}$ (resp., $c'^{(l)}$) denote the random variable having the distribution of the information gathered by the adversary in the $(i+1)$th event in an authentic (resp., simulated) computation, given that the adversary view up to the $i$th event is $\mu_i$. Then, $c^{(l)}$ and $c'^{(l)}$ are equally distributed.* □

## 4.4 Asynchronous verifiable secret sharing

We define Verifiable Secret Sharing in an asynchronous setting (AVSS), and describe an AVSS scheme. Our AVSS scheme is resilient against $t$-limited (Byzantine) adversaries in a network of $n$ parties, as long as $n \geq 4t + 1$.

Our construction uses ideas appearing in [BGW, FM, Fe]. In particular, [Fe] and [CR] describe different AVSS schemes, for $n \geq 4t + 1$ and $n \geq 3t + 1$, respectively (the [CR] scheme is presented in Chapter 5). However, in those schemes the parties have a small probability of error in reconstructing the secret (and in [CR] also a small probability of not terminating), whereas our scheme has no probability of error.

We describe our AVSS scheme as a preamble for our construction for Byzantine adversaries. In our construction we do not use the AVSS scheme as such: in an AVSS scheme, the dealer shares a secret among the parties; later, the parties reconstruct the secret from its shares. In our protocol, the parties' shares of the secret are further processed, and the reconstructed secret is different than the shared secret (much as in the Fail-Stop case). Nevertheless, for clarity and completeness, we first define AVSS and describe a stand-alone AVSS scheme; the components of this scheme will be used in the Byzantine protocol.

In Section 4.4.1 we define AVSS. In Sections 4.4.2 through 4.4.4 we describe our AVSS scheme. In Section 4.4.5 we prove the correctness of our scheme.

### 4.4.1 A definition

A Verifiable Secret Sharing scheme (either synchronous or asynchronous) consists of two protocols: a sharing protocol, in which a dealer shares a secret among the other parties, and a reconstruction protocol, in which the parties reconstruct the secret from its shares. An AVSS scheme should have the following properties: first, a uncorrupted dealer should be able to share a secret in a reconstructible way. Furthermore, if *one* uncorrupted party accepts a sharing of a secret, then *all* the uncorrupted parties accept this sharing, and a unique secret will be reconstructed (even if the dealer is corrupted). Finally, if the dealer is uncorrupted, then the shared secret should remain unknown to the corrupted parties, as long as no uncorrupted party has started the reconstruction protocol. The following definition formalizes these requirements in our asynchronous setting.

**Definition 4.18** *Let $(S, R)$ be a pair of protocols such that each uncorrupted party that completes protocol $S$ subsequently invokes protocol $R$ with its local output of protocol $S$ as local input. We say that $(S, R)$ is a* $t$-resilient AVSS scheme *for $n$ parties if the following holds, for every $t$-limited adversary.*

- **Termination.**
  1. *If the dealer is uncorrupted, then every uncorrupted player will eventually complete protocol $S$.*
  2. *If some uncorrupted party has completed protocol $S$, then all the uncorrupted parties will eventually complete protocol $S$.*
  3. *If a uncorrupted party has completed protocol $S$, then it will complete protocol $R$.*
- **Correctness.** *Once the first uncorrupted party has completed protocol $S$, a* unique *value, $r$, is fixed, such that:*
  1. *$r$ is each uncorrupted party's output of prtocol $R$.*
  2. *If the dealer is uncorrupted, sharing a secret $s$, then $r = s$.*
- **Secrecy.** *If the dealer is uncorrupted and as long as no uncorrupted party has invoked protocol $R$, then the adversary view of the computation is distributed independently of the shared secret.*

**Remark.** We stress that a uncorrupted party is not required to complete protocol $S$ in case that the dealer is corrupted. (We do not distinguish between the case where a uncorrupted party did not complete protocol $S$, and the case where a uncorrupted party has completed $S$ unsuccessfully.)

### 4.4.2 An AVSS scheme

We present an outline of our scheme. Roughly speaking, the sharing protocol (denoted V-Share) consists of three stages: first, each party waits to receive its share of the secret from the dealer. Next, the parties jointly try to verify that their shares define a unique secret. Once a party is convinced that a unique secret is defined, it locally computes and outputs a 'corrected share' of the secret (using the information gathered in the verification stage).

Our AVSS scheme has the following additional property. There exists a polynomial $p(\cdot)$ of degree $t$ such that each uncorrupted party $P_i$'s output of protocol V-Share is $p(i)$, and $p(0)$ is the shared secret. This property is at the heart of our construction for Byzantine adversaries.

In the reconstruction protocol (denoted V-Recon), each party sends its share to the parties in some predefined set $R$ (the set $R$ is an external parameter with the same role as in the Fail-Stop case). Next, each party in $R$ waits to receive enough shares to uniquely determine a secret, and outputs the reconstructed secret. In order to deal with possibly erroneous shares, we use a procedure for error correcting of Generalized Reed-Solomon (GRS) codes in an 'on-line' fashion. This procedure is presented in Section 4.4.4 below.

We turn to describing the V-Share protocol in more detail. To share a secret, $s$, the dealer chooses at random a polynomial $h(\cdot,\cdot)$ of degree $t$ in two variables such that $h(0,0) = s$,[14] and sends each party $P_i$ the $t$-degree polynomials $f_i(\cdot) \triangleq h(i,\cdot)$ and $g_i(\cdot) \triangleq h(\cdot,i)$. Each party $P_i$ now sends a verification message $v_{i,j} = f_i(j)$ to each party $P_j$. If the verification message sent from $P_i$ to $P_j$ is correct, then party $P_j$ has $v_{i,j} = f_i(j) = h(i,j) = g_j(i)$; in this case $P_j$ 'confirms party $P_i$' by Broadcasting $(\mathrm{OK}, j, i)$. Party $P_i$ accepts the shared secret when it finds a large enough set of parties that have confirmed each other in a 'dense' enough manner, described below.

We describe each party's view of the 'OK' Broadcasts in terms of a graph. Namely, let the $\mathrm{OK}_i$ graph be the (undirected) graph over the nodes $[n]$, where an edge $(j,k)$ exists if party $P_i$ completed both the $(\mathrm{OK}, j, k)$ Broadcast (initiated by $P_j$), and the $(\mathrm{OK}, k, j)$ Broadcast (initiated by $P_j$). Define an $(n,t)$-star in a graph:[15]

**Definition 4.19** *Let $G$ be a graph over the nodes $[n]$. We say that a pair $(C,D)$ of sets such that $C \subseteq D \subseteq [n]$ is an $(n,t)$-star in $G$, if the following hold:*

- $|C| \geq n - 2t$
- $|D| \geq n - t$
- *for every $j \in C$ and every $k \in D$ the edge $(j,k)$ exists in $G$.*

In the sequel, we use star to shorthand $(n,t)$-star. Party $P_i$ accepts a shared secret when it finds a star in its $\mathrm{OK}_i$ graph. Lemma 4.26 on page 82 below implies that, provided that $n \geq 4t+1$, the shares of the uncorrupted parties in a star in the $\mathrm{OK}_i$ graph define a unique polynomial of degree $t$ in two variables; Lemma 4.27 implies that the polynomials defined by the stars of every two parties are equal. Thus, once a uncorrupted party finds a star, a unique secret is defined.

Remark: conceptually, we could have let a party wait to have a clique of size $n-t$ (instead of a star) in its OK graph, before accepting a shared secret: if the dealer is uncorrupted, then the OK graph will eventually contain such a clique. However, finding a maximum size clique in a graph is an NP-complete problem. Instead, the party will try to find a star. In Section 4.4.3 we describe an efficient procedure for finding a $(n,t)$-star in a graph, provided that the graph contains a clique of size $n-t$.

We want to make sure that if a uncorrupted party finds a star in its OK graph, then all the uncorrupted parties will find a star, even when the dealer is corrupted and the OK graph does not contain a clique. For this purpose, upon finding a star, the party sends it to all the other parties. Upon receiving an $(\mathrm{OK},\cdot,\cdot)$ Broadcast, a party that has not found a star checks whether any of the suggested stars is indeed a star in its OK graph. Note that an edge in the OK graph of a uncorrupted party will eventually be an edge in the OK graph of every uncorrupted party (since all the $(\mathrm{OK},\cdot,\cdot)$ messages are Broadcasted); thus, a star in the OK graph of some uncorrupted party will eventually be a star in the OK graph of every other uncorrupted party.

[14] That is, $h(x,y) = \sum_{i=0}^{t}\sum_{j=0}^{t} h_{i,j}x^iy^j$, where $h_{0,0} = s$, and all the other coefficients $h_{0,1},\ldots,h_{t,t}$ are chosen uniformly and independently over $F$.

[15] Our definition of an $(n,t)$-star is different than the "standard" definition.

Party $P_i$'s output of protocol V-Share will be $h_i(0,i)$, where $h_i(\cdot,\cdot)$ is the (unique) polynomial defined by the star found by $P_i$. If $P_i$ is a member of its own star, then the polynomial $g_i(\cdot)$ that $P_i$ received from the dealer satisfies $g_i(\cdot) = h_i(\cdot, i)$, and $P_i$ outputs $g_i(0)$. If $P_i$ is not a member of its own star, then $g_i(\cdot)$ may be erroneous (possibly, $P_i$ didn't even receive $g_i(\cdot)$.) Therefore, upon finding a star $(C_i, D_i)$, and if $P_i \notin D_i$, then $P_i$ computes $h_i(0,i)$ using the verification messages $v_{j,i}$ it has received (or still expects to receive) from the parties $P_j \in D_i$. (For the $2t+1$ *uncorrupted* parties $P_j \in D_i$, we have $v_{j,i} = h_i(j,i)$. Therefore, the values received from the parties in $D_i$ uniquely determine the polynomial $h_i(\cdot, i)$.) In Section 4.4.4 we describe how this computation is carried out.

The code of the V-Share and V-Recon protocols is described in Figures 4-9 and 4-10, respectively.

**protocol V-Share**

<u>Code for the dealer (on input $s$):</u>

1. choose a random polynomial $h(\cdot,\cdot)$ of degree $t$ in two variables, such that $h(0,0) = s$. For every $1 \le i \le n$, send the polynomials $f_i(\cdot) = h(\cdot, i)$ and $g_i(\cdot) = h(i,\cdot)$ to party $P_i$.

<u>Code for party $P_i$:</u>

2. Upon receiving $f_i(\cdot)$ and $g_i(\cdot)$ from the dealer, and for each $1 \le j \le n$, send $f_i(j)$ to party $P_j$.
3. Upon receiving $v_{i,j}$ from party $P_j$: if $v_{i,j} = g_i(j)$, then broadcast $(\mathrm{OK}, i, j)$.
4. Upon receiving a broadcast $(\mathrm{OK}, j, k)$, check for the existence of a star in $\mathrm{OK}_i$, using procedure STAR described in Section 4.4.3 below. If a star $(C_i, D_i)$ is found, go to Step 6 and send $(C_i, D_i)$ to all the parties.
5. Upon receiving a message $(C_j, D_j)$ add $(C_j, D_j)$ to the set of 'suggested stars'. As long as a star is not yet found, then whenever an $(\mathrm{OK}, k, l)$ Broadcast is received, check whether $(C_j, D_j)$ form a star in the $\mathrm{OK}_i$ graph.
6. Upon finding a star $(C_i, D_i)$, and if $i \notin D_i$, correct polynomial $g_i(\cdot)$, based on the verification messages received from the parties in $D_i$, and using the error correcting procedure OEC described in Section 4.4.4.

   Namely: let $\mathcal{V}_i = \{(j, v_{i,j}) | j \in D_i\}$; set $g_i(\cdot) = \mathrm{OEC}[t,t](\mathcal{V}_i)$.
7. Once $g_i(\cdot)$ is corrected, (locally) output $g_i(0)$.

Figure 4-9: V-Share - The verifiable sharing protocol

### 4.4.3 Efficiently finding a star

We describe an efficient procedure for finding a $(n,t)$-star in a graph of $n$ nodes (see Definition 4.19 on page 75), provided that the graph contains a clique of size $n-t$. Namely, our procedure outputs either a star in the graph, or a message 'star not found'; whenever the input graph contains a clique of size $n-t$, then the procedure outputs a star in the graph.

We follow an idea of Gabril, appearing in [GJ] p. 134. There, the following approxi-

**Protocol V-Recon**$[R](a_i)$

Code for party $P_i$ (on input $a_i$, and with parameter $R \subseteq \{P_1, \ldots, P_n\}$):

7. Send $a_i$ to the parties in $R$.
8. Let $\mathcal{S}_i = \{(j, a_j) |\ a_j$ has been received from $P_j\}$.
   If $P_i \in R$, terminate without output.
   Otherwise, set $z_i(\cdot)$ =OEC$[t,t](\mathcal{S}_i)$, and output $z_i(0)$.

Figure 4-10: V-Recon - The verifiable reconstruction protocol

mation algorithm is described: if the input graph contains a clique of size $n-k$, then a clique of size $n-2k$ is found. The algorithm is simple: find a **maximal matching**[16] in the **complementary graph**[17] and output the set of unmatched nodes. Clearly, the output is an independent set in the complementary graph; thus, it forms a clique in the original graph. Furthermore, if a graph contains a clique of size $n-k$, then any maximal matching in the complementary graph involves at most $k$ edges and $2k$ nodes.

We first restate our problem in terms of the complementary graph: if the input graph has an independent set of size $n-t$, find sets $C \subseteq D$ of nodes, such that $|C| \geq n-2t$, $|D| \geq n-t$, and no edges exist between nodes in $C$ and nodes in $C \cup D$. We call such a pair of sets an $(n,t)$-$\overline{\text{star}}$. In the rest of this section, we refer to the complementary graph only.

In our procedure, we find a *maximum* matching in the graph (say, using [Ed] or [MV]). [18] Based on this matching, we compute sets $C, D$ of nodes, and check whether $(C, D)$ form an $(n,t)$-$\overline{\text{star}}$ in the graph. If the input graph contains an independent set of size $n-t$, then the computed sets $C, D$ form an $(n,t)$-$\overline{\text{star}}$ in the input graph.

We describe how sets $C$ and $D$ are computed. Consider a matching; we say that a node is a **triangle-head** if it is unmatched, and two of its neighbours are a matched pair (namely, the edge between these two neighbours is in the matching). Let $C$ denote the set of unmatched nodes that are not triangle-heads. Let $B$ be the set of matched nodes that have neighbours in $C$, and let $D \triangleq [n] - B$.

Our procedure is described in Figure 4-11. Figure 4-12 illustrates the relations among the different subsets of nodes.

**Proposition 4.20** *Assume procedure* STAR *outputs* $(C, D)$*, on input graph* $G$*. Then,* $(C, D)$ *form a* $t$-$\overline{\text{star}}$ *in* $G$*.*

**Proof:** Clearly, if algorithm STAR outputs $(C, D)$ then $|C| \geq n-2t$ and $|D| \geq n-t$, and $C \subseteq D$. We show that for every $i \in C$ and every $j \in D$, the nodes $i$ and $j$ are not neighbours in $G$.

[16] A matching in a graph is a set $M$ of edges such that no two edges in $M$ have a common endpoint. A matching is maximal if every edge added to it has a common endpoint with an edge in the matching.

[17] The complementary graph is the graph in which an edge exists iff it does not exist in the original graph.

[18] In fact, we only need that the matching cannot be improved by augmenting paths of length at most 3.

**Procedure STAR**$[t](G)$

<u>input</u>: an undirected graph $G$ (over the nodes $[n]$), a parameter $t$.
<u>output</u>: a $t$-$\overline{\text{star}}$ in the graph $G$, or a message: 'star not found'.

1. Find a maximum matching $M$ in $G$.
   Let $N$ be the set of matched nodes (namely, the endpoints of the edges in $M$), and let $\bar{N} \triangleq [n] - N$.
2. Verify that the matching, $M$, has property $\mathcal{P}$:
   (a) Let $T$ be the set of triangle-heads; namely, set
   $T \triangleq \{i \in \bar{N} | \ \exists j, k \text{ s.t } (j,k) \in M \text{ and } (i,j),(i,k) \in G\}$.
   Let $C \triangleq \bar{N} - T$.
   (b) Let $B$ be the set of matched nodes that have neighbours in $C$;
   namely, let $B \triangleq \{j \in N | \ \exists i \in C \text{ s.t. } (i,j) \in G\}$.
   Let $D \triangleq [n] - B$.
   (c) If $|C| \geq n - 2t$ and $|D| \geq n - t$, output $(C, D)$. Otherwise, output 'star not found'.

Figure 4-11: STAR - A procedure for finding a $\overline{\text{star}}$ in a graph

Figure 4-12: Partition of the graph $G$

Assume that $i \in C$ and $j \in D$, and that $(i, j)$ is an edge in $G$. As $j \in D$, we must have $j \notin B$. By The definition of $B$, we have $j \notin N$ (if $i \in C$ and $j \in N$, then $j \in B$). Furthermore, $i \in C \subseteq \bar{N}$. Thus, both $i$ and $j$ are unmatched. Consequently, the edge $(i, j)$ can be added to the matching to create a larger matching, and the matching is not maximum (in this case, it is not even maximal). □

**Proposition 4.21** *Let $G$ be a graph over $n$ nodes containing an independent set of size $n - t$. Then procedure* STAR *outputs a $t$-$\overline{\text{star}}$ in $G$.*

**Proof:** We show that if the input graph $G$ contains an independent set of size $n - t$, then the sets $C$ and $D$ determined in Steps 2(a) and 2(b) of procedure STAR are large enough (i.e., $|C| \geq 2t + 1$ and $|D| \geq n - t$). Consequently, the procedure outputs $(C, D)$ in Step 4; Proposition 4.20 assures us that $(C, D)$ form a star in $G$.

First, we show that $|C| \geq n - 2t$. Let $I \subseteq [n]$ be an independent set of size $n - t$ in $G$, and let $\bar{I} = [n] - I$. We adopt the definitions of $N$, $T$, and $C$ in the procedure (see Figure 4-12).

Let $F \triangleq I - C$. We show below that $|F| \leq |\bar{I}|$. However, $|\bar{I}| \leq t$. Consequently, we have $|C| \geq |I| - |F| \geq n - 2t$.

To see that $|F| \leq |\bar{I}|$, we show a one-to-one correspondence $\phi : F \to \bar{I}$. Let $i \in F$; since $i \notin C$, we have either $i \in N$ or $i \in T$.

**Case 1:** $i \in N$. Then, let $\phi(i)$ be the node matched to $i$ in $M$. Clearly, $\phi(i) \in \bar{I}$: otherwise, we had an edge $(i, \phi(i))$ where both $i$ and $\phi(i)$ are in an independent set.

**Case 2:** $i \in T$. By the definition of $T$, node $i$ has two neighbours $j, k$ such that $(j, k) \in M$. Arbitrarily set $\phi(i) = j$. Clearly, both $j$ and $k$ are in $\bar{I}$.

Wo show that $\phi$ is one-to-one. Consider two distinct nodes $l, m \in F$; we distinguish three case:

**Case 1:** $l, m \in N$. In this case, $\phi(l) \neq \phi(m)$ since $M$ is a matching.

**Case 2:** $l \in N$ and $m \in T$. Since $m \in T$, there exists an edge between $m$ and the node matched to $\phi(m)$. Since $l \in N$, the node matched to $\phi(l)$ is $l$. Now, assume that $\phi(l) = \phi(m)$. Thus, $(l, m)$ is an edge in $G$. However, both $l$ and $m$ are in the independent set $I$: a contradiction.

**Case 3:** $l, m \in T$. Assume $\phi(l) = \phi(m)$. Let $a$ be the node matched to $\phi(m)$ in $M$; then, both $l$ and $m$ are neighbours of both $\phi(m)$ and $a$. However, in this case the matching $M$ is not maximum since, for instance, $M - \{(\phi(m), a)\} \cup \{(\phi(m), l), (a, m)\}$ is a larger matching.

It remains to show that $|D| \geq n - t$. Recall that $D = [n] - B$. We show below that $|B| \leq |M|$; since $G$ contains an independent set of size $n - t$, we have $|M| \leq t$. Thus, $|D| = n - |B| \geq n - |M| \geq n - t$.

To see that $|B| \leq |M|$, we show that at most one of the endpoints of every edge $(a, b) \in M$ is in $B$. Suppose on the contrary that both $a$ and $b$ have neighbours in $C$, and let $c, d \in C$ be the neighbours of $a$ and $b$, respectively. Surely, $c \neq d$ (otherwise, $c$ was a triangle-head, and we had $c \notin C$). However, in this case the matching $M$ is not maximum, since, for instance, $M - \{(a, b)\} \cup \{(a, c), (b, d)\}$ is a larger matching. □

### 4.4.4 On-line error correcting

Consider the following scenario. A party expects to receive messages from $m$ parties (message $a_j$ from party $P_j$), so that there exists a polynomial $p(\cdot)$ of degree $d$, satisfying $p(j) = a_j$ for every message $a_j$. The messages arrive one by one, in some arbitrary order; furthermore, up to $t$ of the messages may be wrong or missing. (In our context, we have $m = n - t$ and $d = t$.) We describe an efficient procedure that enables the party to compute this polynomial 'on-line'; namely, the party will recognize when the received messages define a unique 'interpolated polynomial' of degree $d$, and compute this polynomial. The following definitions provide tools for precisely stating this problem.

**Definition 4.22** *Let $\delta$ and $\rho$ be integers, and let $S \subseteq [n] \times F$ such that for every two elements $(i,a)$ and $(i',a')$ in $S$ we have $i \neq i'$. We say that $S$ is $(\delta,\rho)$-*interpolated *if there exists a polynomial $p(\cdot)$ of degree $\delta$, so that at most $\rho$ elements $(i,a) \in S$ do* not *satisfy $p(a) = e$. We say that $p(\cdot)$ is a $(\delta,\rho)$-*interpolated polynomial *of $S$.*

(Remark: A $d$-interpolated vector, defined in Section 4.3.3 on page 61, is a different formulation of a $(d,0)$-interpolated set.) Note that the $(\delta,\rho)$-interpolated polynomial of a $(\delta,\rho)$-interpolated set $S$ is unique, provided that $|S| \geq \delta + 2\rho + 1$. (Proof: assume $S$ has two $(\delta,\rho)$-interpolated polynomials $p(\cdot), q(\cdot)$. Then, for at least $|S| - 2\rho$ elements $(a,e) \in S$ we have $p(a) = e = q(a)$. However, $|S| - 2\rho \geq \delta + 1$; thus, $p(\cdot) = q(\cdot)$.)

**Definition 4.23** *Let $\mathcal{I}$ be an accumulative set [19]. We say that $\mathcal{I}$ is* eventually $(\delta,\tau)$-interpolated, *if for every $\tau$-limited adversary, and every run, there exists an integer $0 \leq \rho \leq \tau$, such that $\mathcal{I}$ will eventually hold a $(\delta,\rho)$-interpolated set of size at least $\delta + \tau + \rho + 1$.*

Let $\mathcal{I}$ be an eventually $(\delta,\tau)$-interpolated accumulative set. Using a similar argument to the one used above, it can be seen that all the $(\delta,\rho)$-interpolated sets of size at least $\delta + \tau + \rho + 1$ held in $\mathcal{I}$ have the same interpolated polynomial. We call this polynomial the $(\delta,\tau)$-interpolated polynomial of $\mathcal{I}$.

Using these notations, the (dynamic) input of the procedure described in this section is an eventually $(d,t)$-interpolated accumulative set $\mathcal{I}$. The required output of this procedure is the $(d,t)$-interpolated polynomial of $\mathcal{I}$. (Definition 4.23 assures us that at least $d+t+1$ values in $\mathcal{I}$ will 'sit on a polynomial' of degree $t$. At least $t+1$ of thus values originate with uncorrupted parties. Consequently, the $(d,t)$-interpolated polynomial of $\mathcal{I}$ is bound to be the 'correct' polynomial, namely the polynomial defined by the values of the uncorrupted parties.)

Our procedure, denoted OEC (for On-line Error Correcting), consists of up to $t$ iterations. In iteration $r$, the party waits until the accumulative set $\mathcal{I}$ is of size at least $d + t + r + 1$; then, the party uses a procedure, described below, that determines whether this set is $(d,r)$-interpolated, and computes the corresponding interpolated polynomial. If a $(d,r)$-interpolated polynomial is found, then we output this polynomial and terminate. Otherwise, we proceed to iteration $r+1$ (since $\mathcal{I}$ is $(d,t)$-eventually interpolated, it is bound to have at least one more element; thus, iteration $r+1$ will be completed).

[19] Accumulative sets are defined in Section 4.1.3 on page 53.

It is left to describe how to determine if a given set is $(d,r)$-interpolated, and how to compute the interpolated polynomial. We use a result from Coding theory. Consider the following code: a word $W = (i_1,a_1)\ldots(i_l,a_l)$ is a code-word iff there exists a polynomial $p(\cdot)$ of degree $d$ such that $p(i_j) = a_j$ for every $1 \le j \le l$. This is a Generalized Reed-Solomon (GRS) code; GRS codes have an efficient error correcting procedure that detects and corrects up to $r$ errors in an input word, $W$, provided that $|W| \ge d + 2r + 1$ (see, for instance, [MS pp. 294-315]). Let EC denote such a procedure. Namely, let $S$ be a $(d,r)$-interpolated set of size at least $d + 2r + 1$; then, procedure EC, on input $(d,r,S)$, outputs the $(d,r)$-interpolated polynomial of $S$.

Procedure OEC is described in Figure 4-13.

**Procedure OEC**$[d,t](\mathcal{I})$

For $0 \le r \le t$ do:

1. Let $I_r$ denote the contents of accumulative set $\mathcal{I}$, when $\mathcal{I}$ contains $d+t+r+1$ elements.

   Wait until $|\mathcal{I}| \ge d+t+r+1$. Then, run EC$(d,r,I_r)$, and let $p(\cdot)$ be the output polynomial.

   If $p(\cdot)$ is a $(d,r)$-interpolated polynomial of $I_r$ (namely, if for $d+t+1$ elements $(i,a) \in I_r$ we have $p(i) = a$), output $p(\cdot)$. Otherwise, proceed to the next iteration.

Figure 4-13: OEC - A procedure for on-line error correcting

**Proposition 4.24** *Let $\mathcal{I}$ be an eventually $(d,t)$-interpolated accumulative set. Then, procedure OEC$[d,t](\mathcal{I})$ halts and outputs the $(d,t)$-interpolated polynomial of $\mathcal{I}$.*

**Proof:** Let $\hat{r}$ be the smallest $r$ such that $I_{\hat{r}}$ is $(d,\hat{r})$-interpolated; since $\mathcal{I}$ is eventually $(d,t)$-interpolated, we have $\hat{r} \le t$. All the iterations up to iteration $\hat{r}$ will be completed (unsuccessfully). The $(d,t)$-interpolated polynomial of $\mathcal{I}$ will be found in iteration $\hat{r}$. □

### 4.4.5 Correctness of the AVSS scheme

**Theorem 4.25** *The pair (V-Share, V-Recon) is a $t$-resilient AVSS scheme in a network of $n$ parties, provided that $n \ge 4t + 1$.*

**Proof:** We assert the Correctness, Termination and Secrecy requirements of Definition 4.18 on page 74.

**Correctness.** We associate with every uncorrupted party $P_i$ that completed protocol V-Share a unique polynomial $h_i(\cdot,\cdot)$ of degree $t$ in two variables, and show that every two uncorrupted parties $P_i$ and $P_j$ have $h_i(\cdot,\cdot) = h_j(\cdot,\cdot)$. Next, we show that Conditions 1 and 2 of the Correctness requirement are met, with respect to $r = h_i(0,0)$ (for some uncorrupted party $P_i$). Moreover, we show that party $P_i$'s output of protocol V-Share is $h_i(i,0)$.

We use two technical lemmas.

**Lemma 4.26** *Let $m \geq d+1$, and let $f_1(\cdot)\ldots f_m(\cdot)$ and $g_1(\cdot)\ldots g_m(\cdot)$ be polynomials of degree $d$ over a field $F$ with $|F| \geq m$, such that for every $1 \leq i \leq d+1$ and every $1 \leq j \leq m$ we have $f_i(j) = g_j(i)$ and $g_i(j) = f_j(i)$. Then, there exists a unique polynomial $h(\cdot,\cdot)$ of degree $d$ in two variables so that for every $1 \leq i \leq m$ we have $h(\cdot,i) = f_i(\cdot)$ and $h(i,\cdot) = g_i(\cdot)$.*

**Proof:** See Appendix B. □

**Lemma 4.27** *Let $h(\cdot,\cdot)$, $h'(\cdot,\cdot)$ be two polynomials of degree $d$ in two variables over a field $F$ with $|F| > d$, and let $v_1 \ldots v_{d+1}$ be distinct elements in $F$. Assume that for every $1 \leq i,j \leq d+1$ we have $h(v_i,v_j) = h'(v_i,v_j)$. Then, $h(\cdot,\cdot) = h'(\cdot,\cdot)$.*

**Proof:** See Appendix B. □

Let $P_i$ be a uncorrupted party that completed protocol V-Share, and let $(C_i, D_i)$ be the star found by $P_i$. Let $D_i'$ be the set of uncorrupted parties in $D_i$, and let $C_i'$ be the set of uncorrupted parties in $C_i$; thus, $|D_i'| \geq |D| - t \geq n - 2t$ and $|C_i'| \geq |C| - t \geq n - 3t \geq t+1$ (since $n \geq 4t+1$). Applying Lemma 4.26, we get that the polynomials $f_j(\cdot), g_j(\cdot)$ of the parties $j \in D_i'$ determine a unique polynomial of degree $t$ in two variables. Let $h_i(\cdot,\cdot)$ denote this polynomial. (Namely, $h_i(\cdot,\cdot)$ is the polynomial associated with $P_i$.) Note that $h_i(\cdot,\cdot)$ is fixed once $P_i$ has completed protocol V-Share.

For every other uncorrupted party $P_j$, let $I_{i,j}$ be the set of uncorrupted parties in $D_i \cap D_j$. Since $n \geq 4t+1$, we have $|D_i \cap D_j| \geq n - 2t \geq 2t+1$, thus $|I_{i,j}| \geq t+1$. For every two parties $k,l \in I_{i,j}$ we have $h_i(k,l) = v_{k,l} = h_j(k,l)$, where $v_{k,l}$ is the verification piece sent by $P_k$ to $P_l$ in Step 2 of protocol V-Share. Applying Lemma 4.27, we have $h_i(\cdot,\cdot) = h_j(\cdot,\cdot)$. The value $r$ required in the Correctness condition is $r = h_i(0,0)$.

We assert Condition 1 (namely, that if the dealer is uncorrupted and has shared a value $s$, then $r = s$). If the dealer is uncorrupted and has chosen a polynomial $h(\cdot,\cdot)$ in Step 1, then for every two parties $P_k, P_l \in D_i'$ we have $h_i(k,l) = h(k,l)$. Applying Lemma 4.27 again, we get $h_i(\cdot,\cdot) = h(\cdot,\cdot)$. In the sequel, we omit the subscript from the polynomial $h(\cdot,\cdot)$.

Next, we show that each uncorrupted party $P_i$'s output of protocol V-Share is $h(i,0)$. Polynomial $h(i,\cdot)$ is the (only) interpolated polynomial of $P_i$'s accumulative set $\mathcal{V}_i$ in Step 6. Therefore, the output of the error correcting procedure OEC in Step 6 will be $h(i,\cdot)$, and the output of protocol V-Share will be $h(i,0)$.

It remains to assert Condition 2 (namely, that $P_i$'s output of V-Recon is $r$). Every uncorrupted party $P_j$ will Broadcast $h(j,0)$ in Step 7; thus, $h(\cdot,0)$ is the (only) interpolated polynomial of $P_i$'s accumulative set $\mathcal{S}_i$ in Step 8. Therefore, the output of the error correcting procedure OEC in Step 8 will be $h(\cdot,0)$, and the output of protocol V-Recon will be $h(0,0) = r$.

**Termination.** Condition 1. If the dealer is uncorrupted, then for every two uncorrupted parties $P_j$ and $P_k$, both $(\mathrm{OK},j,k)$ and $(\mathrm{OK},k,j)$ will be broadcasted, since $f_j(k) = h(k,j) = g_k(j)$ and $g_j(k) = h(j,k) = f_k(j)$. Thus, every uncorrupted party $P_i$ will eventually have a clique of size $n-t$ in its $\mathrm{OK}_i$ graph. Therefore, procedure STAR will find a star in $\mathrm{OK}_i$ and Step 4 will be completed. Step 6 will be completed since the input of procedure OEC (namely, the accumulative set $\mathcal{V}_i$ which is based on the star found in Steps 4 or 5) is eventually $(t,t)$-interpolated.

Condition 2. Let $P_i$ be a uncorrupted party that completed protocol V-Share, and let $(C_i, D_i)$ be the star found by $P_i$. Then, $(C_i, D_i)$ will eventually be a star in the $\mathrm{OK}_j$

graph of every uncorrupted party $P_j$, unless $P_j$ has already completed protocol V-Share. Furthermore, party $P_j$ will receive the $(C_i, D_i)$ message (sent by $P_i$ in Step 4), and will verify, in Step 5, that the sets $(C_i, D_i)$ form a star in $OK_j$. Upon finding a star, $P_j$ will execute Step 6 and complete protocol V-Share.

Condition 3. If all the uncorrupted parties have started protocol V-Recon, then the accumulative set $\mathcal{S}_i$ of Step 8 of each uncorrupted party $P_i$ is eventually $(t,t)$-interpolated. Thus, all the uncorrupted parties will complete procedures OEC and V-Recon.

**Secrecy.** We use the following notations.

- For a value $v$, let $\mathcal{H}_v$ denote the set of polynomials of degree $t$ in two variables, with free coefficient $v$.
- We say that a sequence $f_1(\cdot), \ldots, f_t(\cdot), g_1(\cdot), \ldots, g_t(\cdot)$ of polynomials is interleaved if for every $1 \leq i, j \leq t$ we have $f_i(j) = g_j(i)$. Let $I$ denote the set of interleaved sequences of $2t$ polynomials of degree $t$.

**Lemma 4.28** *Let $F$ be a field with $|F| > d$, and let $s \in F$. Then, for every interleaved sequence $f_1(\cdot), \ldots, f_d(\cdot), g_1(\cdot), \ldots, g_d(\cdot)$ in $I$, there exists a unique polynomial $h(\cdot,\cdot) \in \mathcal{H}_s$, so that for every $1 \leq i \leq d$ we have $h(\cdot, i) = f_i(\cdot)$ and $h(i, \cdot) = g_i(\cdot)$.*

**Proof:** See Appendix B. □

Assume a uncorrupted dealer, and let $s$ be the shared value. Then, the dealer has chosen, in Step 1 of protocol V-Share, a polynomial $h(\cdot,\cdot)$ with uniform distribution over $\mathcal{H}_s$. Furthermore, all the relevant information a set of $t$ parties received during an execution of protocol V-Share, is an interleaved sequence $f_1(\cdot), \ldots, f_t(\cdot), g_1(\cdot), \ldots, g_t(\cdot)$ in $I$, so that for every $1 \leq i \leq t$ we have $h(\cdot, i) = f_i(\cdot)$ and $h(i, \cdot) = g_i(\cdot)$.

Lemma 4.28 implies that for every shared value $s \in F$, this correspondence between polynomials in $\mathcal{H}_s$ and interleaved sequences in $I$ is one to one and onto. Therefore, a uniform distribution over the polynomials in $\mathcal{H}_s$ induces a uniform distribution over the interleaved sequences in $I$.

Thus, all the corrupted parties have after executing protocol V-Share is a sequence of interleaved polynomials of degree $t$, chosen with uniform distribution over $I$, regardless of the shared value. □

## 4.5 Byzantine adversaries

As in the Fail-Stop case, let the computed function be $f : F^n \to F$, and assume that the parties have an arithmetic circuit computing $f$. We describe an $n$ party protocol for securely $t$-computing $f$ in an asynchronous network with arbitrary (i.e., Byzantine) adversaries, provided that $n \geq 4t + 1$.

We follow the outline of the Fail-Stop protocol, modifying its components to the Byzantine setting. First, we extend Shamir's Secret Sharing scheme (used for Fail-Stop adversaries) to an Asynchronous Verifiable Secret Sharing (AVSS) scheme. Next, The multiplication step is adapted to a Byzantine setting.

In our protocol we do not use the AVSS scheme as such: in an AVSS scheme, the dealer shares a secret among the parties; later, the parties reconstruct the secret from its shares. In our protocol, the parties' shares of the secret are further processed, and the reconstructed secret is different than the shared secret (much as in the Fail-Stop case). Nevertheless, for clarity and completeness, we first define AVSS and describe a stand-alone AVSS scheme; the components of this scheme will be used in the Byzantine protocol.

Our AVSS scheme, as well as the multiplication step protocol, makes use of error correcting techniques for Generalized Reed-Solomon (GRS) codes. In the AVSS scheme we describe a procedure, run locally by each party, that enables the party to locate and correct, in an 'on-line' fashion, erroneous or missing messages in a sequence of received messages. In the multiplication step, a GRS code-word is generated, so that each party holds a piece of this code-word. Each party shares its piece, using AVSS; then, the parties agree, without learning further information, on a set of parties whose shared pieces are indeed the pieces of the original code-word.

### 4.5.1 Global Verifiable Share

The Global Verifiable Share (GV-Share) protocol, described in Figure 4-14, is the Byzantine counterpart of the Fail-Stop GShare protocol (described in Figure 4-4 on page 60). It consists of two phases: first, each party shares its input, using the V-Share protocol of the AVSS scheme (Figure 4-9 on page 76); next, the parties use protocol ACS (Section 4.2.3 on page 56) to agree on a set $C$ of at least $n-t$ parties who properly shared their inputs. Party $P_i$'s output of protocol GV-Share is this set $C$, and the $i$th share of each secret shared by a party in $C$. (Recall that protocol GShare had an additional security parameter, $d$; here, the security parameter is fixed to $d = t$.)

**Protocol GV-Share**($x_i$)

Code for Party $P_i$, on input $x_i$:

1. Initiate V-Share$_i(x_i)$ (with $P_i$ as the dealer).
   For $1 \le j \le n$, participate in V-Share$_j$.
   Let $v_j$ be the output of V-Share$_j$.
2. Let $\mathcal{U}_i = \{j |$ V-Share$_j$ has been completed$\}$. Set $C$ =ACS$[n-t, n](\mathcal{U}_i)$.
3. Once the set $C$ is computed, output $(C, \{v_j | j \in C\})$.

Figure 4-14: GV-Share - The global verifiable sharing protocol

### 4.5.2 Computing a multiplication gate

Let $c = a \cdot b$ be a multiplication gate, and let $A(\cdot), B(\cdot)$ be the polynomials associated with the input lines. Namely, each uncorrupted party $P_i$'s shares of these lines are $A(i)$ and $B(i)$ respectively, and $A(0) = a$ and $B(0) = b$. As in the Fail-Stop case, the parties will jointly compute their shares of a random polynomial $C(\cdot)$ of degree $t$ with $C(0) = A(0) \cdot B(0)$, so that each uncorrupted party $P_i$'s share of the output line will be $C(i)$.

The Byzantine multiplication procedure follows the outline of its Fail-Stop counterpart. Namely, the parties first generate a random polynomial $D(\cdot)$ of degree $2t$ with free coefficient

$D(0) = A(0) \cdot B(0)$. Then, the parties compute their shares of the truncation polynomial of $D(\cdot)$ to degree $t$; this truncation polynomial is the output polynomial $C(\cdot)$.

We proceed to describe the Byzantine implementations of these two steps.

### Randomization

The Byzantine randomization step follows the outline of its Fail-Stop counterpart. Namely, each party $P_i$ first shares (in a way described below) a random polynomial $H_i(\cdot)$ of degree $2t$ with $H_i(0) = 0$. Next, the parties use protocol ACS to agree on a set $C$ of parties that have successfully shared their polynomial. Finally, each party $P_i$ locally computes $H(i) = \sum_{j \in C} H_j(i)$, and $D(i) = A(i) \cdot B(i) + H(i)$.

It remains to describe how each party $P_i$ shares its polynomial $H_i(\cdot)$. We use the [BGW] method. This method 'has the effect' of sharing polynomial $H_i(\cdot)$ in the 'straightforward' way. Namely, on one hand, each party $P_j$ will have $H_i(j)$; on the other hand, the information gathered by the corrupted parties will be 'equivalent' to each corrupted party knowing only its share of $H_i(\cdot)$. Consequently, the information gathered by the corrupted parties *in the entire multiplication step* will be independent of the computed value (i.e., $A(0) \cdot B(0)$).

We describe this sharing method. Each party $P_i$ shares $t$ uniformly chosen values, using $t$ invocations of V-Share. Let $z_{i,j,k}$ be party $P_k$'s output of the $j$th invocation of V-Share where $P_i$ is the dealer. Upon completing all the $t$ invocations of V-Share, each party $P_k$ locally computes $H_i(k) = \sum_{j=1}^{t} k^j \cdot z_{i,j,k}$.

Let us reason this slightly unintuitive sharing method (for a formal proof, see the proof of Theorem 4.30 on page 90). Let $S_{i,j}(\cdot)$ be the polynomial of degree $t$ defined by the $j$th V-Share initiated by $P_i$ (namely, $S_{i,j}(k) = z_{i,j,k}$ for every uncorrupted party $P_k$). Polynomial $H_i(\cdot)$ is now defined as $H_i(x) = \sum_{j=1}^{t} x^j \cdot S_{i,j}(x)$: each party $P_k$ locally computes $H_i(k) = \sum_{j=1}^{t} k^j \cdot S_{i,j}(k) = \sum_{j=1}^{t} k^j \cdot z_{i,j,k}$. Let $s_{i,j,l}$ be the coefficient of $x^l$ in $S_{i,j}(x)$; it might be helpful to visualize

$$
\begin{array}{lcccccccccccc}
\multicolumn{13}{l}{H_i(x) =} \\
s_{i,1,0}x & + & s_{i,1,1}x^2 & + & \dots & + & s_{i,1,t-1}x^t & + & s_{i,1,t}x^{t+1} & + & & & \\
 & & s_{i,2,0}x^2 & + & \dots & + & s_{1,2,t-2}x^t & + & s_{i,2,t-1}x^{t+1} & + & s_{i,2,t}x^{t+2} & + & \\
 & & & & \dots & & & & & & & & \\
 & & & & & & s_{i,t,0}x^t & + & s_{i,t,1}x^{t+1} & + & \dots & + & s_{i,t,t}x^{2t}
\end{array}
$$

The free coefficient of $H_i(\cdot)$ is 0; thus, $H_i(0) = 0$. Each polynomial $S_{i,j}(\cdot)$ is of degree $t$; thus, $H_i(x)$ is of degree $2t$. Furthermore, it can be seen that the coefficients of the monomials $x, \dots, x^t$ in $H_i(x)$ (and, thus, the same coefficients of the sum polynomial $H(\cdot)$ and in polynomial $D(\cdot)$) are uniformly distributed over $F$. Consequently, the coefficients of all the non-zero powers of the truncation polynomial $C(\cdot)$) are uniformly distributed over $F$.

Clearly, the corrupted parties gather some extra information on top of the $t$ shares of $H_i(\cdot)$. However, it is plausible that this information is independent of $A(0) \cdot B(0)$: party $P_i$ chooses $t^2 + t$ random coefficients, and the corrupted parties receive only $t^2$ values. (In the proof of Lemma 4.31 on page 91 we show that the information gathered by the corrupted parties during the whole multiplication protocol is independent of $A(0) \cdot B(0)$.)

**Degree-reduction**

Next, the parties use their shares of polynomial $D(\cdot)$ in order to jointly and securely compute their shares of the 'truncation' of $D(\cdot)$ to degree $t$; namely, the $t+1$ coefficients of the output polynomial $C(\cdot)$ are the coefficients of the $t+1$ lower degrees of $D(\cdot)$.

In the Fail-Stop protocol, the parties computed their shares of $C(\cdot)$ by invoking a protocol for 'multiplying the inputs vector by a fixed matrix'; we shortly review this protocol. Let $\vec{d} = D(1), \ldots, D(n)$, let $\vec{c} = C(1), \ldots, C(n)$, and let $M$ be the $n \times n$ matrix such that $\vec{d} \cdot M = \vec{c}$. First, each party $P_i$ shared its 'input' $D(i)$; next, the parties agreed on a set of parties that have successfully shared their inputs. Once this set, $G$, was agreed upon, each party locally computed the appropriate matrix $M^G$, and the products of his shares by $M^G$. Finally the parties invoked $n$ Reconstruct protocols, so that in the $i$th invocation of Reconstruct, party $P_i$ computed $C(i) = \sum_{j \in G} d_j \cdot M^G_{j,i}$.

In the Byzantine setting, the parties will use V-Share and V-Recon instead of the simple sharing scheme of the Fail-Stop case. Still, a major problem remains: the agreed set $G$ may contain (corrupted) parties $P_i$ that have shared some value different than the expected value $D(i)$; in this case, the parties will not have the expected outputs. In the rest of this section, we describe how the parties make sure that the value associated with each party $P_i$ in the agreed set (namely, the free coefficient of the $t$-degree polynomial defined by the uncorrupted parties' shares of $P_i$'s input) is indeed $D(i)$.

For a party $P_i$, let $s_i$ be the value associated with $P_i$ (recall that $s_i$ is fixed once the first uncorrupted party has completed $P_i$'s V-Share); for a set $A$ of parties, let $S_A = \{(i, s_i) | P_i \in A\}$. We first note that it is enough to agree on a set $G$ of at least $3t+1$ parties, such that $S_G$ is $(2t, 0)$-interpolated [20] (namely, all the values shared by the parties in $G$ 'sit on a polynomial of degree $2t$'). This is so, since the set $G$, being of size $3t+1$, contains at least $2t+1$ *uncorrupted* parties; thus, the interpolated polynomial of $S_G$ is bound to be $D(\cdot)$.

We describe a protocol for agreement on a set $A$ of parties, such that $S_A$ is $(2t, 0)$-interpolated. This protocol, denoted AIS (for Agreement on an Interpolated Set), is a 'distributed implementation' of procedure OEC (described in Section 4.4.4 on page 80). Protocol AIS consists of up to $t$ iterations. In iteration $r$ $(0 \leq r \leq t)$, the parties first use protocol ACS (Figure 4-3 on page 58) to agree on a set, $G_r$, of at least $3t+1+r$ parties that have successfully shares their inputs. Next, the parties perform a computation, described below, to check whether $S_{G_r}$ is $(2t, r)$-interpolated. If $S_{G_r}$ is $(2t, r)$-interpolated, then there exists a set $G'_r \subseteq G_r$, of size at least $3t+1$, such that $S_{G'_r}$ is $(2t, 0)$-interpolated; the parties will compute and output $G'_r$. Otherwise (i.e., $S_{G_r}$ is not $(2t, r)$-interpolated), the parties will proceed to the next iteration. We stress that the parties will not know the interpolated polynomial of each $S_{G_r}$. They will only know whether such a polynomial exists.

It remains to describe how to check, given a set $G$ of size $3t+1+r$, whether $S_G$ is $(2t, r)$-interpolated, and how to compute the corresponding set $G'$ (i.e., $G' \subseteq G$, $|G'| \geq 3t+1$, and $S_{G'}$ is $(2t, 0)$-interpolated). As in procedure OEC, we use error correcting for Generalized Reed-Solomon codes. However, in procedure OEC, the 'word', $S_G$, was a (dynamic) input of one party. Thus, each party could locally run an error correcting procedure. In our setting, each party has only one share of each element of $S_G$; the parties will invoke a joint computation implementing a specific error correcting procedure, and use it to check whether

[20] Interpolated sets and interpolated polynomials were defined in Section 4.4.4 on page 80.

$G$ is $(2t, r)$-interpolated and to compute $G'$.

Let us first outline the particular error correcting procedure we will be implementing. The inputs to this procedure are $(d, r, W)$. If $|W| \geq d + 2r + 1$ and $W$ is $(d, r)$-interpolated, then the output is the interpolated polynomial of $W$. (Otherwise, an appropriate message is output). The procedure consists of three steps:

**Computing the syndrome.** For an input word $W = (i_1, s_1) \ldots (i_l, s_l)$, let $V_{l \times l}$ be the (Vandermonde) matrix defined by $V_{j,k} = (i_k)^j$. Let $\vec{a} = s_1 \ldots s_l$. The **syndrome** of $W$ is the $l - (d+1)$ right most elements of the $l$-vector product $\vec{a} \cdot V^{-1}$.
First, compute the syndrome of $W$.

**Remark.** Let $Q(\cdot)$ be the polynomial so that for every $(i, a) \in W$ we have $Q(i) = a$. Then, the vector $\vec{a} \cdot V^{-1}$ is the vector of the coefficients of $Q(\cdot)$; the syndrome consists of the coefficients of the monomials $x^{d+1}, \ldots, x^l$ of $Q(x)$. In particular, if $Q(\cdot)$ is of degree $d$ (namely, $W$ is a code-word), then the syndrome is the zero vector.

**Computing the error-vector.** The **error-vector** is the $l$-vector $\vec{e} = e_1 \ldots e_l$, where $e_j$ is the 'displacement of $s_j$ from the correct value'. Namely, assume that $W$ is $(d, r)$-interpolated, and let $P(\cdot)$ be the $(d, r)$-interpolated polynomial of $W$; then $e_j = P(i_j) - s_j$, for every element $(i_j, s_j) \in W$. The error-vector is unique, since the interpolated polynomial $P(\cdot)$ is unique.

Compute the error-vector, using the syndrome. A widely used implementation of this step is the Berlkamp-Massey algorithm (see [MS pp. 365-368]).

**Remark.** We stress that the error-vector can be computed based on the syndrome *only*. If the input word, $W$, is not $(d, r)$-interpolated, then the computed error-vector may be erroneous.

**Computing the output polynomial.** Choose $2t + 1$ correct elements in $W$ (namely, elements $(i_j, a_j)$ such that $e_j = 0$), and use them to interpolate $P(\cdot)$. (This step will not be implemented.)

An important observation is that the syndrome can be represented as a function of the error vector only; thus, it holds no information on the $(d, r)$-interpolated polynomial, $P(\cdot)$, of $W$. Namely, let $\vec{P}$ (resp. $\vec{Q}$) be the coefficients vector of the polynomial $P(\cdot)$ (resp. $Q(\cdot)$), completed to length $l$. (Polynomial $Q(\cdot)$ is the polynomial satisfying $Q(i) = a$ for every pair $(i, a) \in W$.) Then,

$$\vec{Q} = \vec{a} \cdot V^{-1} = (\vec{P} \cdot V + \vec{e}) \cdot V^{-1} = \vec{P} + \vec{e} \cdot V^{-1}.$$

The last $l - (d+1)$ elements in $\vec{P}$ are zero. Therefore, for $l - (d+1) < i \leq l$, we have $Q_i = [\vec{e} \cdot V]_i$. Consequently, the last $l - (d+1)$ elements in $\vec{Q}$ (namely, the syndrome) are a linear combination of the elements of $\vec{e}$ only.

Let us now describe how we implement and use this error correcting procedure in our setting. Each element of the syndrome is a linear combination of the inputs; thus, the parties can jointly compute the syndrome. That is, given an agreed set, $G$, each element of the syndrome of $S_G$ is computed as follows. Each party computes the appropriate linear combination of its shares, and invokes a V-Recon protocol with the result of this linear combination as input. Once all these V-Recon protocols are completed, each party has the full syndrome of $S_G$.

Once the syndrome is computed, each party uses the Berlkamp-Massey algorithm in order to *locally* compute the error vector. If, in iteration $r$, $S_{G_r}$ is $(2t, r)$-interpolated, then the computed error vector is the 'true' error vector of $S_{G_r}$; however, if $S_{G_r}$ is not $(2t, r)$-interpolated, then the computed error vector may be incorrect. Consequently, the parties draw the following conclusions. (Each party draws these conclusions locally; still, all the uncorrupted parties draw the same conclusions.) If the computed error vector, denoted $\vec{e}'$, contains more than $r$ non-zero elements, then surely $S_{G_r}$ is not $(2t, r)$-interpolated, and the parties proceed to the next iteration. If $\vec{e}'$ contains up to $r$ non-zero elements, then the parties still have to verify that $\vec{e}'$ is the correct error vector: let $G'_r$ be the set of parties $P_i$ such that $e'_i = 0$; the parties will recompute the syndrome, based on $G'_r$ alone. If the recomputed syndrome is all zeros, then $S_{G'_r}$ is $(2t, 0)$-interpolated, and the parties output $G'_r$ and terminate. If the recomputed syndrome is non-zero, the parties conclude that $S_{G_r}$ is not $(2t, r)$-interpolated, and proceed to the next iteration. [21]

Protocol AIS is described in Figure 4-15. Let $z_{i,j}$ be $P_i$'s share of the value shared by $P_j$. The (dynamic) input of each party $P_i$ is the following accumulative set, denoted $\mathcal{Z}_i$: once $P_i$ has successfully completed $P_j$'s V-Share, the pair $(j, z_{i,j})$ is added to $\mathcal{Z}_i$. The parties' common output is a set $G$ of at least $3t + 1$ parties, such that each uncorrupted party $P_i$ has completed the V-Share of every party in $G$ (namely, $G \subseteq \{P_j | (j, z_{i,j}) \in \mathcal{Z}_i\}$), and $S_G$ is $(2t, 0)$-interpolated.

**Claim 4.29** *Assume that protocol AIS is run with dynamic inputs $\mathcal{Z}_1, \ldots, \mathcal{Z}_n$ as described above. Then all the uncorrupted parties terminate protocol AIS with a common set G of at least $3t + 1$ parties, such that $S_G$ is $(2t, 0)$-interpolated.*

**Proof:** First, assume that the ACS protocol of Step 1 of some iteration is completed by the uncorrupted parties. Then, all the uncorrupted parties compute the same syndrome in Step 2 of this iteration; thus, all the uncorrupted parties make the same decisions in Step 3. Consequently, all the uncorrupted parties complete this iteration; furthermore, either all the parties output a $(2t, 0)$-interpolated set of size at least $3t + 1$, or all the parties proceed to the next iteration.

Next, we show, by induction on the number of iterations, that as long as the required set is not found, all the uncorrupted parties will complete each invocation of ACS. For the base of induction, we note that the sequence $\mathcal{U}_i, \ldots, \mathcal{U}_n$ of the accumulative sets defined in Step 1 of the first iteration is $(3t + 1, n)$-uniform (see Definition 3.6 on page 37). Thus, all the uncorrupted parties will complete the ACS invocation of the first iteration. For the induction step, assume that the uncorrupted parties have completed the ACS protocol of iteration $r$. Consequently, the sequence $\mathcal{U}_1, \ldots, \mathcal{U}_n$ is $(3t + r + 1, t)$-uniform. Assume further that the set agreed upon in iteration $r$ is not $(2t, r)$-interpolated; let $G_r$ denote this set. Then, $G_r$ contain at most $3t$ *uncorrupted* parties. Thus, there exists at least one uncorrupted party in $[n] - G_r$. This uncorrupted party will eventually be in the accumulative sets of all the uncorrupted parties; thus the collection $\mathcal{U}_1 \ldots \mathcal{U}_n$ is $(3t + r + 2, t)$-uniform. Consequently, all the uncorrupted parties will complete the ACS invocation of iteration $r + 1$.

[21] A more "time-efficient" version of this protocol lets the parties execute all the $t$ iterations 'in parallel'; consequently, the running time of the protocol is the running time of the slowest iteration. (In this 'parallel' version, different parties may complete the 'iterations' in different order. Thus, the parties need to execute an additional simple protocol to agree on the iteration whose output is adopted.)

**Protocol AIS($\mathcal{Z}_i$)**

Code for party $P_i$, on dynamic input $\mathcal{Z}_i$.
for $0 \leq r \leq t$ do

1. Let $\mathcal{U}_i = \{P_j | (j, z_{i,j}) \in \mathcal{Z}_i\}$.
   Set $G =$ACS$[3t + 1 + r, n](\mathcal{U}_i)$.
2. Once $G$ is computed, compute the syndrome of $S_G$:
   Let $V$ be the Vandermonde matrix of the indices in $G$. Namely, let $G = k_1 \ldots k_{|G|}$; then, $V_{i,j} = (k_j)^i$.
   Let $\vec{z_i} = z_{i,k_1}, \ldots, z_{i,k_{|G|}}$.
   For $2t + 1 < j \leq |G|$ set $\sigma_j =$V-Recon$([\vec{z_i} \cdot V^{-1}]_j, [n])$.
   Let $\vec{\sigma} = \sigma_1 \ldots \sigma_{t+r}$. ($\vec{\sigma}$ is the syndrome of $S_G$).
3. Run the Berlkamp-Massey algorithm on $\vec{\sigma}$, and let $\vec{e'}$ be the output.
   (a) If $\vec{e'}$ has more than $r$ non-zero elements, continue to the next iteration ($S_G$ is not $(2t, r)$-interpolated).
   (b) If $\vec{e'}$ has up to $r$ non-zero elements, verify that $\vec{e'}$ is correct:
   Let $G'$ be the set of parties in $G$ whose corresponding entry in $\vec{e'}$ is zero. Repeat step 2 with respect to $G'$.
   If the syndrome of $S_{G'}$ is the zero vector, output $G'$ and halt.
   Otherwise, proceed to the next iteration ($S_G$ is not $(2t, r)$-interpolated).

Figure 4-15: AIS - The agreement on an interpolated set protocol

Finally, we note that $G_t$ (namely, the set $[n]$) is $(2t, t)$-interpolated; thus, the required set will be found in iteration $t$, unless it was found beforehand. □

### The multiplication protocol

Combining the Randomization and the Degree-Reduction steps, we derive a protocol for computing a multiplication gate. The code is presented in Figure 4-16.

**Protocol BMUL**

Code for party $P_i$, on inputs $a_i$ and $b_i$.
**Randomization**

1. For $1 \leq k \leq t$, execute V-Share$_{i,k}(r_k)$, where $r_k \in_R F$ and $P_i$ is the dealer.
   For $1 \leq j \leq n$, for $1 \leq k \leq t$, participate in V-Share$_{j,k}$.
   Let $h_{i,j,k}$ be $P_i$'s output of V-Share$_{j,k}$.
2. Let $\mathcal{U}_i = \{P_j |$ V-Share$_{j,k}$ has completed, for all $1 \leq k \leq n\}$.
   Set $G =$ACS$[n-t, n](\mathcal{U}_i)$.
   Set $h_i = \sum_{j \in G} \sum_{k=1}^{t} i^k \cdot h_{i,j,k}$, and $d_i = a_i \cdot b_i + h_i$.

**Degree-Reduction**

3. Once $d_i$ is computed, execute V-Share$_i(d_i)$, where $P_i$ is the dealer.
   For $1 \leq j \leq n$, participate in V-Share$_j$.
4. Let $z_{i,j}$ be $P_i$'s share of $P_j$'s shared secret,
   and let $\mathcal{Z}'_i = \{(j, z_{i,j}) |$ V-Share$_j$ has been completed$\}$.
   Set $G' =$AIS$(\mathcal{Z}'_i)$.
5. Let $V^{\hat{G}'}$ be the matrix used in the Fail-Stop multiplication step, as described in section 4.3.3 on page 61, and let $\vec{z_i} = z_{i,j_1} \ldots z_{i,j_{|G'|}}$, where $j_1 \ldots j_{|G'|}$ are the indices of the parties in $G'$.
   For $1 \leq j \leq n$, Set $c_j =$V-Recon$([\vec{z}^{i}_{\hat{G}} \cdot (V^{\hat{G}'}]_j, \{j\})$.
6. Once $c_i$ is computed, output $c_i$.

Figure 4-16: BMUL - The Byzantine multiplication gate protocol

## 4.5.3 The Byzantine protocol

The overall structure of the Byzantine protocol is the same as that of the Fail-Stop protocol. Namely, in the Byzantine protocol, denoted Bcompute, the parties execute the code of protocol FStop, with the exception that protocols GShare, MUL, and Reconstruct are replaced by protocols GV-Share, BMUL, and V-Recon, respectively.

**Theorem 4.30** *Let $f : F^n \rightarrow F$ for some field $F$ with $|F| > n$, and let $A$ be a circuit computing $f$. Then, protocol Bcompute$[A]$ asynchronously $(\lceil \frac{n}{4} \rceil - 1)$-securely computes $f$ in the bounded secure channels setting in the presence of adaptive adversaries and non-erasing parties.*

**Proof:** Here we consider only the case of non-adaptive adversaries. The case of adaptive adversaries is proven in the same way as in Section 4.3.6.

The proof of the Termination property, as well as the construction of the ideal-model adversary is the same as in the proof of Theorem 4.13 on page 63 (security of protocol FS-Compute). We stress that our construction of the ideal-model adversary for Fail-Stop adversaries is valid even for corrupted parties that change their inputs, although in the Fail-Stop case the inputs of the corrupted parties are not changed.

The validity of this construction in the Byzantine setting is shown in the same way as in the Fail-Stop case, with the exception that lemmas analogous to Lemmas 4.15 and 4.14 (on pages 68 and 67, respectively) need to be proven. The proof of the Byzantine analogue of Lemma 4.15 is similar to the proof of Lemma 4.15, with the exception that now, the correctness of the uncorrupted parties' outputs of the *Byzantine* implementations of the different protocols (namely, GV-Share, V-Recon, and BMUL) need to be proven. Correctness of the GV-Share and V-Recon protocols is implied by the correctness of the AVSS scheme; correctness of protocol BMUL is discussed in Section 4.5.2.

In order to state the Byzantine analogue of Lemma 4.14, let us redefine some notations. The view of a set of parties is defined as in the proof of Theorem 4.13. Fix an adversary and an input vector $\vec{x}$. The random variables $\mu$ (resp. $\mu'$) take the distribution of the corrupted parties' view of protocol Bcompute, run on input $\vec{x}$ (resp. the corrupted parties' view of a simulated computation). Lemma 4.31 below is the Byzantine analogue of Lemma 4.14. Our proof of Theorem 4.30 is thus completed. □

**Lemma 4.31** *Fix an adversay and an input vector. Then, the corresponding random variables $\mu$ and $\mu'$ are identically distributed.*

**Proof:** We use the notations of Lemma 4.14. Namely, fix a prefix, $V_i$, of length $i$ of a view (namely, $V_i$ is a prefix of an instance of either $\mu$ or $\mu'$). The random variable $\hat{s}_{i+1}$ (resp. $\hat{s}'_{i+1}$) describes the distribution of the $(i+1)$th delivered message in $\mu$ (resp. $\mu'$), given that $V_i$ is the corresponding prefix of the view.

As in the proof of Lemma 4.14, it is enough to show that the *contents* of $\hat{s}_{i+1}$ and $\hat{s}'_{i+1}$ are identically distributed, for the case that the sender of $\hat{s}_{i+1}$ is uncorrupted and the recipient is corrupted. Each message of protocol Bcompute falls into one of the following cases.

**Messages of some invocation of V-Share.** We distinguish two cases: if the dealer of the relevant invocation of V-Share is uncorrupted, then equality of the distributions of $\hat{s}_{i+1}$ and $\hat{s}'_{i+1}$ is implied by the proof of the privacy property of the AVSS scheme (Theorem 4.25 on page 81). If the dealer is corrupted, then the contents of $\hat{s}_{i+1}$ and $\hat{s}'_{i+1}$ can be inferred from the common prefix $V_i$ (and are, thus, equal).

**Messages of some invocation of ACS.** This case is trivial, since messages of protocol ACS have empty contents.

The last three cases are different types of invocations of protocol V-Recon. We note that, as in the Fail-Stop case, in order to show that the contents of *all* the messages of some invocation of V-Recon are identically distributed in the two computations, it suffices to show that the *output* of this invocation of V-Recon is identically distributed in the two computations.

**Messages of the final invocation of V-Recon.** (Namely, Step 4 of protocol Bcompute. See figure 4-8 on page 64.) This case is shown using similar considerations to those of the Fail-Stop case.

**Messages of an invocation of V-Recons within protocol AIS.** (Namely, Step 2 of the AIS.) See Figure 4-15 on page 89.) We have seen in Section 4.5.2 that the outputs of the V-Recons of the AIS protocol are the different syndromes computed by the parties; furthermore, we have seen that the syndromes are uniquely determined by the errors introduced by the corrupted parties. However, these errors can be inferred from the common prefix $V_i$: the values associated with each corrupted party, $P_i$, in each set $G_r$ can be inferred from the messages sent by $P_i$ in the corresponding invocation of V-Share (these messages appear in the common prefix $V_i$). Thus, the errors introduced by the corrupted parties, as well as the different syndromes, are identical in the two computations.

**Messages of an invocation of V-Recon within protocol BMUL.** (Namely, Step 5 of BMUL. See Figure 4-16 on page 90.) As in the Fail-Stop case, we show that the corrupted parties' outputs of the $t$ invocations of V-Recon, along with the contents of the other messages of this invocation of BMUL, received by the corrupted parties, are uniformly and independently distributed, regardless of the values of the input lines of the corresponding multiplication gate.

The polynomial $H(x)$ interpolated by the parties in Step 2 (Randomization) of protocol BMUL is $H(x) = \sum_{i \in G} \sum_{j=1}^{t} x^j \cdot S_{i,j}(x)$. (The set $G$ is the output of the corresponding invocation of ACS, and each polynomial $S_{i,j}(\cdot)$ is the polynomial defined by the uncorrupted parties' outputs of $P_i$'s $j$th invocation of V-Share. See Section 4.5.2 on page 85 for more details.) Let us first regard this polynomial in a more convenient way: reversing the order of summation, we have $H(x) = \sum_{j=1}^{t} x^j \cdot \hat{S}_j(x)$, where $\hat{S}_j(\cdot) = \sum_{i \in G} S_{i,j}(\cdot)$. All the coefficients of each polynomial $\hat{S}_j(\cdot)$ are uniformly distributed, since the set $G$ contains uncorrupted parties. The contents of the randomization-phase messages received by the corrupted parties constitute $t$ shares of each polynomial $\hat{S}_j(\cdot)$. Consequently, the data gathered by the corrupted parties during the entire BMUL protocol adds up to $t$ shares of each one of the $t$ polynomials $\hat{S}_1(\cdot), \ldots, \hat{S}_t(\cdot)$, along with the outputs of the $t$ invocations of V-Recon (namely, $t$ shares of the truncation polynomial $C(\cdot)$).

Fix some arbitrary input polynomials $A(\cdot)$ and $B(\cdot)$ of the multiplication step. Lemma 4.32 below shows that for each sequence of $t^2 + t$ field elements gathered by the corrupted parties there exists a unique choice of the polynomials $\hat{S}_1(\cdot) \ldots \hat{S}_t(\cdot)$ that yield this sequence. However, the polynomials $\hat{S}_1(\cdot) \ldots \hat{S}_t(\cdot)$ are uniformly distributed; thus, the sequence of field-elements gathered by the corrupted parties is uniformly distributed, both in $\mu$ and in $\mu'$. [22]

□

[22] In Lemma 4.32 we assume, without loss of generality, that the corrupted parties are $P_1, \ldots, P_t$.

**Lemma 4.32** *Let $F$ be a finite field with $|F| > d$, and let $A(\cdot)$ and $B(\cdot)$ be polynomials of degree $d$ over $F$. Then, for every sequence $a_{1,1}, \ldots, a_{d,d}, c_1, \ldots, c_d$ of elements in $F$ there exists a unique sequence $\hat{S}_1(\cdot), \ldots, \hat{S}_t(\cdot)$ of polynomials of degree $d$ such that:*

1. *Fore each $1 \leq i, j \leq d$, we have $\hat{S}_i(j) = a_{i,j}$.*
2. *Let $C(\cdot)$ be the truncation to degree $d$ of the polynomial*

$$D(x) = A(x) \cdot B(x) + \sum_{j=1}^{d} x^j \cdot \hat{S}_j(x)$$

*Then, for $1 \leq i \leq d$ we have $C(i) = c_i$.*

**Proof:** See Section 4-B. □

## 4.6 Lower bounds

We show that our protocols have optimal resilience, both in the Fail-Stop and in the Byzantine cases. Firsr we show, in Theorem 4.34, that there exist functions that cannot be $\lceil \frac{n}{3} \rceil$-securely computed in an asynchronous network of $n$ parties if Fail-Stop adversaries are allowed. We prove this result in two steps. First we reduce the problem of secure computation in a *synchronous* network with $n = 2t$ to the problem of secure computation in an *asynchronous* network with $n = 3t$. Next we use the results of [BGW] (and also Chor and Kushilevitz [CK]) that there exist functions that cannot be securely computed (or even approximated) when $n \leq 2t$. Simple examples of such functions are the OR and AND functions of $n$ boolean inputs. Even though a direct (and conceivably shorter) impossiblity proof for the asynchronous case is possible, we believe that our proof by reduction is clearer as well as more general.

Next we show, in Theorem 4.35, that there exist functions that cannot be $\lceil \frac{n}{4} \rceil$-securely computed with no probability of error in an asynchronous network if general (Byzantine) adversaries are allowed. This proof is a generalization of a technique used in [BGW]. Both Theorems 4.34 and 4.35 apply even to non-adaptive adversaries.

We remark that Ben-Or Kelmer and T. Rabin [BKR] show, using techniques from [BGW, CCD, CR], how any function can be asynchronously $(\lceil \frac{n}{3} \rceil - 1)$-securely computed, in the presence of Byzantine adversaries, with exponentially small (but positive) probability of either not terminating or having wrong outputs.

For the proof, we use the following notations. For an asynchronous protocol $\rho$, a set $G$ of parties, an adversary $\mathcal{A}$ and input $\vec{x}$, let $\mu_{\rho,G,\mathcal{A}}(\vec{x})$ be the random variable having the distribution of the view of the parties in $G$ when running protocol $\rho$ with adversary $\mathcal{A}$ and input $\vec{x}$. (Unlike the notion of adversary view defined in previous sections, here the view of a set of parties does not include the information seen by the scheduler.)

A party's output of a computation is a function of its view only. Consequently, if $\mu_{\rho,G,\mathcal{A}}(\vec{x})$ and $\mu_{\rho,G,\mathcal{A}'}(\vec{x})$ are identically distributed (for some protocol $\rho$, input $\vec{x}$, some set $G$ of parties, two adversaries $\mathcal{A}$ and $\mathcal{A}'$ and some input $\vec{x}$), then the *output* of the parties in $G$ is identically distributed with adversaries $\mathcal{A}$ and $\mathcal{A}'$.

### 4.6.1 Fail-Stop adversaries

We use the following result of [CK]. This result refers to *synchronous* networks where the adversaries are **eavesdropping** (namely, the corrupted parties only try to gather information, but otherwise follow the protocol). In this model, we say that a protocol **approximates** a boolean function, if for every input, with probability greater than $\frac{1}{2}$ the uncorrupted parties output the function value. Informally, a protocol is $t$-**private** if every adversary gathers no information from executing the protocol, other than the output of the uncorrupted parties (namely the computed function value).

**Theorem 4.33 [BGW, CK]:** *For every $n \geq 2$, there exist boolean functions $f$ such that there is no synchronous $\lceil \frac{n}{2} \rceil$-private protocol for $n$ parties that approximates $f$.*

Some intuition for the proof of Theorem 4.33 follows. Consider the case where $n = 2$, and $f$ is the OR function of two boolean inputs. The first party, $A$, should make sure that if the input of the other party, $B$, is 0 then the communication between the parties will be independent of $A$'s input (otherwise $B$ will learn about $A$'s input). Similarly, $B$ should make sure that if the input of $A$ is 0 then the communication between the parties will be independent of $B$'s input. Now, consider the case where both $A$ and $B$ have input 0. Since the communication must be independent of both inputs the parties will never be able to decide on an output.

This argument can be generalized in a simple way to deal with three parties in an asynchronous network with one possible Fail-stop fault. Assume that $A$ is good, and the third party, $C$, does not respond to $A$'s messages. Then, $A$ must complete the protocol based only on its communication with $B$, since $C$ may be faulty. Furthermore, the communication with $B$ should be independent of $A$'s input (for the case where $C$ is good but slow and $B$ is faulty with input 0). The argument now continues as in the synchropbous case.

To be more precise, we prove the following lower bound based on Theorem 4.33.

**Theorem 4.34** *For every $n \geq 3$, there exist boolean functions $f$, such that no asynchronous protocol for $n$ parties securely $\lceil \frac{n}{3} \rceil$-computes $f$ when Fail-Stop adversaries are allowed.*

**Proof:** Let $n \geq 3$, and let $m = n - \lceil \frac{n}{3} \rceil$. (Hence, $m = \lfloor \frac{2n}{3} \rfloor$ and $\lceil \frac{m}{2} \rceil \leq \lceil \frac{n}{3} \rceil$.) Let $f' : \{0,1\}^m \rightarrow \{0,1\}$ be a boolean function as in Theorem 4.33 (namely, $f'$ cannot be approximated by any $\lceil \frac{m}{2} \rceil$-private protocol). Construct the function $f : \{0,1\}^n \rightarrow \{0,1\}$ so that for every $n$-vector $\vec{x}$, we have $f(\vec{x}) = f'(\vec{x}_{[m]})$. We show that *if* there exists a protocol, $\pi$, that securely $\lceil \frac{n}{3} \rceil$-computes $f$ in an asynchronous network of $n$ parties with Fail-Stop adversaries, *then* there exists an $\lceil \frac{m}{2} \rceil$-private *synchronous* protocol, $\pi'$, for $m$ parties, that approximates $f'$. The theorem follows.

Suppose we have an $n$-party asynchronous protocol, $\pi$, that securely $\lceil \frac{n}{3} \rceil$-computes $f$. We construct an $m$-party synchronous protocol, $\pi'$, as follows. In protocol $\pi'$ each party will simulate a party executing protocol $\pi$ *in an asynchronous network of $n$ parties.* The simulation proceeds as follows. Let $P'_1, \ldots, P'_m$ be the parties of the actual synchronous network. Let $P_1, \ldots P_n$ be the 'virtual' parties addressed by protocol $\pi$. Actual party $P'_i$ will simulate virtual party $P_i$; parties $P_{m+1} \ldots P_n$ are not simulated. Each (actual) party keeps a queue of incoming messages. In each (synchronous) communication round, the party invokes a cycle of protocol $\pi$ for the first message in the queue. Whenever protocol

$\pi$ instructs to send a message to 'virtual' party $P_i$ for $i \in [m]$, the party sends this message to $P'_i$. If $i > m$, the instruction is ignored. Once virtual party $P_i$ terminates, actual party $P'_i$ terminates with the output of protocol $P_i$.

We note that if the asynchronous protocol, $\pi$, doesn't terminate, then the synchronous protocol $\pi'$ doesn't terminate as well. Thus, there may exist executions in which protocol $\pi'$ doesn't terminate. We describe how we avoid this phenomenon at the end of the proof.

We assert the validity of protocol $\pi'$ (namely, we show that it is $\lceil \frac{m}{2} \rceil$-private and approximates $f$). An outline of the proof follows. For every *synchronous* adversary interacting with protocol $\pi'$, we describe three *synchronous* adversaries, denoted $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, for protocol $\pi$:

- Adversary $\mathcal{A}_1$ is the adversary that corresponds to the virtual execution of protocol $\pi$, when run by protocol $\pi'$. With this adversary, the messages sent by $t$ parties re delayed (these are the parties $P_{m+1} \ldots P_n$), and additional $t$ parties are corrupted (these are the parties that correspond to the actual corrupted parties).

- Adversary $\mathcal{A}_2$ delivers the messages in the same order as adversary $\mathcal{A}_1$. However, the parties $P_1 \ldots P_m$ are uncorrupted, and the parties $P_{m+1} \ldots P_n$ are corrupted. The security of protocol $\pi$ asserts that with this adversary the parties $P_1 \ldots P_m$ output the correct function value.

- Adversary $\mathcal{A}_3$ delivers the messages sent by the parties $P_1 \ldots P_m$ in the same order as adversary $\mathcal{A}_1$. The messages of the parties $P_{m+1} \ldots P_n$ are postponed until there are no undelivered messages from the parties $P_1 \ldots P_m$. The corrupted parties are the parties that correspond to the actual corrupted parties. The security of protocol $\pi$ asserts that with this adversary the set of corrupted parties gathers no information, other than the output of the uncorrupted parties.

We show below that, for any synchronous adversary that interacts with $\pi'$, the outputs of the parties $P_1, \ldots, P_m$ running protocol $\pi$ are identically distributed in the presence of all three (asynchronous) adversaries. It follows that, for every (synchronous) adversary the uncorrupted parties output the correct function value, and the corrupted parties gather no information other than the computed function value. Consequently, the synchronous protocol, $\pi'$, is $t$-private and approximates $f$.

We now describe adversaries $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ in more detail. We partition the set of parties of protocol $\pi$ (namely, $P_1, \ldots, P_n$) as follows. Let $B$ be the set of parties that correspond to the corrupted parties the synchronous network (namely, $P_i \in B$ iff $P'_i$ is corrupted). Let $G$ be the set of parties that correspond to *uncorrupted* parties in the synchronous network (i.e.,, $G = \{P_1 \ldots P_m\} - B$). Let $L$ be the set of "silent" parties (namely, $L = \{P_{m+1}, \ldots, P_n\}$).

**Adversary $\mathcal{A}_1$.** <u>Corrupted parties:</u> the corrupted parties are the parties in $B$; they follows the protocol without fail-stopping.
<u>Scheduler:</u> messages sent from the parties in $P_1, \ldots, P_m$ are delivered in a 'round robin'; messages sent from the parties in $L$ are not delivered.

**Adversary $\mathcal{A}_2$.** <u>Corrupted parties:</u> the corrupted parties are the parties in $L$; they do not send any messages.
<u>Scheduler:</u> messages sent from the parties $P_1, \ldots, P_m$ are delivered in a 'round robin'.

By the definition of asynchronous secure computation (Definition 4.4 on page 52), the following property holds with adversary $\mathcal{A}_2$: on every input, $\vec{x}$, the parties in $G \cup B$ must complete protocol $\pi$ with output $f_C(\vec{x})$, for some core set, $C$, of size at least $m = n - \lceil \frac{n}{3} \rceil$. The only inputs available are the inputs of the parties $P_1, \ldots, P_m$. Thus, with adversary $\mathcal{A}_2$, the parties in $G \cup B$ output $f_{[m]}(\vec{x})$. (In the sequel, we use this observation only for the parties in $G$.)

Furthermore, it can be seen, by induction on the number of communication events, that the following relation holds for every asynchronous protocol, $\rho$, and for every input vector, $\vec{x}$: Every prefix of a view of the parties $P_1, \ldots, P_m$ when running protocol $\rho$ on input $\vec{x}$ is identically distributed with adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$. Thus, $\mu_{\pi,[m],\mathcal{A}_1}(\vec{x})$ and $\mu_{\pi,[m],\mathcal{A}_2}(\vec{x})$ are identically distributed. Consequently, the outputs of the parties $P_1, \ldots, P_m$ are identically distributed with adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$.

**Adversary $\mathcal{A}_3$.** Corrupted parties: as with adversary $\mathcal{A}_1$.
Scheduler: messages sent from the parties $P_1, \ldots, P_m$ are delivered in a 'round robin'; messages sent from the parties in $L$ are delayed until all the messages from parties in $P_1, \ldots, P_m$ are delivered.

For every input, $\vec{x}$, let $\mu'_{\pi,[m],\mathcal{A}_3}(\vec{x})$ denote the longest prefix of $\mu_{\pi,[m],\mathcal{A}_3}(\vec{x})$ that does not contain messages sent by parties in $L$. It can be seen (again, by induction on the number of events in a prefix of a view) that the random variables $\mu'_{\pi,[m],\mathcal{A}_3}(\vec{x})$ and $\mu_{\pi,[m],\mathcal{A}_1}(\vec{x})$ are identically distributed; in particular, they have the same support set. We have seen above that in every execution where the view of the parties in $G$ is in the support set of $\mu_{\pi,[m],\mathcal{A}_1}(\vec{x})$, these parties terminate. Consequently, with adversary $\mathcal{A}_3$ the parties in $G$ terminate, deciding on an output, once their view is in the support set of $\mu'_{\pi,[m],\mathcal{A}_3}(\vec{x})$ (i.e., before any message from a party in $L$ is delivered). Furthermore, the parties in $G$ have the same output as with adversary $\mathcal{A}_1$, namely $f_{[m]}(\vec{x})$. In addition, since protocol $\pi$ is secure, adversary $\mathcal{A}_3$ gathers no information other than the output of the uncorrupted parties (namely, $f_{[m]}(\vec{x})$). Consequently, the corrupted parties gather no information other than the computed function value with adversary $\mathcal{A}_1$.

It remains to fix the termination condition of protocol $\pi'$. We use the following provision. If, in the 'simulated execution', protocol $\pi$ doesn't terminate after some predefined large enough number of rounds, then the party terminates with some default output. We compute this limit on the number of rounds: With adversary $\mathcal{A}_1$ protocol $\pi$ terminates with probability 1. Consequently, there exist an integer, $k$, such that on every input, with probability at least $\frac{3}{4}$ all the uncorrupted parties complete protocol $\pi$ after $k$ communication events. (The limit $\frac{3}{4}$ is arbitrary. Any number in the interval $(\frac{1}{2}, 1)$ would do.) We fix the limit on the number of rounds to $k$. Thus, protocol $\pi'$ always terminates, and with probability at least $\frac{3}{4}$ the uncorrupted parties output the output of protocol $\pi$, namely $f_{[m]}(\vec{x}) = f'(\vec{x}_{[m]})$. (We note that this "truncated" synchronous protocol is $\lceil \frac{n}{2} \rceil$-private, since the limit, $k$, is independent of the inputs.) $\square$

### 4.6.2 Byzantine adversaries

For the proof of the lower bound for the Byzantine case we use the following additional notation. Fix some adversary and input vector. Let $A$ be a set of parties. Let the **conversation**, $\mathcal{C}_A$, among the parties in $A$, be the random variable having the distribution of the sequence of all messages sent among the parties in $A$. (The order of the messages in each instance of the conversation is induced by their order in the corresponding view.)

**Theorem 4.35** *For every $n \geq 4$, there exist functions $f$ such that no asynchronous protocol for $n$ parties securely $\lceil \frac{n}{4} \rceil$-computes $f$, if Byzantine adversaries are allowed.*

**Proof (sketch):** First, we note that the straightforward reduction to the synchronous case (used in the previous proof) does not hold in the Byzantine case, for the following reason. In the Fail-Stop case, a uncorrupted party has no way of telling whether a party is uncorrupted or corrupted; therefore, each uncorrupted party must terminate whenever *any* set of $n-t$ parties are ready to terminate. in the presence of Byzantine adversaries, there may exist strategies for the corrupted parties that cause the uncorrupted parties to recognize some party, $P$, as corrupted. In this case, the uncorrupted parties can wait for $n-t$ parties *other than* $P$. Consequently, in the simulation of the previous proof, if some corrupted party is recognized as corrupted then the uncorrupted parties may never terminate.

We prove the Theorem for the case $n = 4$. The proof can be easily generalized to all $n$. Let $\pi$ be a four party protocol that securely 1-computes the following function:

$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_2 = x_3 = 1 \\ 0 & \text{otherwise} \end{cases}$$

Let $P_1$, $P_2$, $P_3$ and $P_4$ be the parties (and let $x_i$ be the input of $P_i$).

Consider the following setting. Party $P_1$ is corrupted (with a strategy described below), and the adversary delivers the messages of parties $P_1$, $P_2$ and $P_3$ in a 'round robin'; messages sent by $P_4$ are delivered only when there are no undelivered messages of the other parties. We show below that, with small but non-zero probability, both $P_2$ and $P_3$ do not recognize $P_1$ as being corrupted. In this case, both $P_2$ and $P_3$ will terminate *before any message of $P_4$ is delivered;* this can be shown using similar considerations to those of the previous proof. We show that in this case, the output of $P_2$ is different than the output of $P_3$.

We first observe that the conversation $\mathcal{C}_{\{P_2,P_3\}}$ is independent of both $x_2$ and $x_3$: consider, for contradiction, the first message in $\mathcal{C}_{\{P_2,P_3\}}$ that depends on the input of its sender (say, $P_2$). Then, $P_3$ can learn $x_2$, regardless of the value of $x_3$. Consequently, it is the combination of the conversations $\mathcal{C}_{\{P_2,P_3\}}$ and $\mathcal{C}_{\{P_2,P_1\}}$ that determines the output of $P_2$. Similarly, it is the combination of the conversations $\mathcal{C}_{\{P_3,P_2\}}$ and $\mathcal{C}_{\{P_3,P_1\}}$ that determines the output of $P_3$.

Now, assume the following strategy of $P_1$: send some random string instead of each message expected of $P_1$. Let $x_2 = x_3 = 1$. With small but non-zero probability, the combination of the conversations $\mathcal{C}_{\{P_1,P_2\}}$ and $\mathcal{C}_{\{P_2,P_3\}}$ is consistent with input 0 of $P_3$ (and some input of $P_1$), and at the same time the combination of the conversations $\mathcal{C}_{\{P_1,P_3\}}$ and $\mathcal{C}_{\{P_3,P_2\}}$ is consistent with input 1 of $P_2$ (and some input of $P_1$). In this case, both $P_2$ and $P_3$ terminate *before any message of $P_4$ is delivered.* Furthermore, $P_2$ outputs 0, and $P_3$ outputs 1. That is, $P_2$ and $P_3$ have different outputs and the protocol is not 1-secure. □

**Remark:** Using a technique similar to the technique of the above proof, it can be shown that there exist functions that cannot be securely computed in a *synchronous* network where a third of the parties are corrupted. (This result is stated in [BGW].) Moreover, the result for the synchronous case is much stronger: consider a secure, synchronous protocol for computing the AND function of three variables (i.e., $\text{AND}(x_1, x_2, x_3) = 1$ iff $x_1 = x_2 = x_3 = 1$) in a network where the parties are $\{P_1, P_2, P_3\}$, and one corrupted party, $P_1$, does not send any messages. Using the same argument as in the above proof, it can be shown that the view of party $P_2$ is independent of the input of $P_3$, and the view of party $P_3$ is independent of the input of $P_2$. Consequently, there must exist inputs for which the output of the uncorrupted parties is incorrect with probability at least one half (instead of negligible probability in the above proof).

We note that this 'synchronous' strategy for the corrupted party is useless in our asynchronous model: if a corrupted party, $P$, does not send messages (or, alternatively, if this party is 'caught sending nonsense messages'), then the uncorrupted parties may interact with $n-t$ parties *other than* $P$.

## 4-A Expected running times

We analyze the running times of protocols FScompute and Bcompute. First, let us informally present the standard definition of the running time of an asynchronous protocol.

Consider a virtual 'external clock' measuring time in the network (of course, the players cannot read this clock). Let the **delay** of a message be the time elapsed from its sending to its receipt. Let the **period** of a finite execution of a protocol be the longest delay of a message in this execution. The **duration** of a finite execution is the total time measured by the global clock divided by the period of this execution. (Infinite executions have infinite duration.)

The **expected running time** of a protocol, is the maximum over all inputs and applicable adversaries, of the average over the random inputs of the players, of the duration of an execution of the protocol.

Let $n$ be the number of parties, and let $d$ be the depth of the computed circuit. Consider protocol FScompute. The expected running time of protocol ACS is $O(\log n)$; thus, the expected running time of protocol GShare is also $O(\log n)$. Protocol Reconstruct runs in constant time. Consequently, each invocation of protocol MUL has expected running time of $O(\log n)$, and Protocol FScompute has expected running time of $O(d \cdot \log n)$.

Consider protocol Bcompute. Protocols V-Share and V-Recon run in constant time. Protocol AIS, as presented in Figure 4-15 on page 89, consists of $O(n)$ iterations, where each iteration has expected running time of $O(\log n)$. However, as remarked at the end of Section 4.5.2, we can let all iterations run 'in parallel'; in this version, protocol AIS has expected running time of $O(\log n)$. Consequently, protocol Bcompute runs in $O(d \cdot \log n)$ expected time.

## 4-B Proofs of technical lemmas

In the sequel, we let $V^{(k)}$ denote the $k \times k$ (Vendermonde) matrix, where $(V^{(k)})_{i,j} = i^j$. Note that $V^{(k)}$ is non-singular. (When the dimension $k$ is obvious, we write $V$ instead of $V^{(k)}$.)

**Lemma 4.3** *Let $F$ be a finite field with $|F| > d$, and let $s \in F$. Then for every sequence $v_1, \ldots, v_d, u_1, \ldots, u_d$ of field-elements, there exists a unique polynomial $p(\cdot)$ of degree $2d$ with $p(0) = s$, such that:*

1. *For $1 \le i \le d$, we have $p(i) = v_i$.*

2. *For $1 \leq i \leq d$, we have $q(i) = u_i$, where $q(\cdot)$ is the truncation of $p(\cdot)$ to degree $d$ (i.e., the coefficients of $q(\cdot)$ are the coefficients of the $d+1$ lower degrees of $p(\cdot)$).*

**Proof:** Let $p_0, \ldots, p_{2d}$ be the coefficients of polynomial $p(\cdot)$ claimed in the Lemma. Then, the following $2d+1$ equations hold:

$$\begin{array}{ccccccccccccc}
p_0 1^0 & + & \ldots & + & p_d 1^d & + & p_{d+1} 1^{d+1} & + & \ldots & + & p_{2d} 1^{2d} & = & v_1 \\
\ldots \\
p_0 d^0 & + & \ldots & + & p_d d^d & + & p_{d+1} d^{d+1} & + & \ldots & + & p_{2d} d^{2d} & = & v_d \\
p_0 & + & \ldots & + & 0 & + & 0 & + & \ldots & + & 0 & = & s \\
p_0 1^0 & + & \ldots & + & p_d 1^d & + & 0 & + & \ldots & + & 0 & = & u_1 \\
\ldots \\
p_0 d^0 & + & \ldots & + & p_d d^d & + & 0 & + & \ldots & + & 0 & = & u_d
\end{array} \tag{4.1}$$

We show that the equations (4.1) (in the variables $p_0, \ldots, p_{2d}$) are linearly independent. The coefficients matrix $M$ of the left hand side of (4.1) can be partitioned into:

$$M_{(2d+1)\times(2d+1)} = \begin{bmatrix} A_{d\times(d+1)} & B_{d\times d} \\ C_{(d+1)\times(d+1)} & 0 \end{bmatrix}$$

Matrix $C$ equals $V^{(d+1)}$; thus the last $d+1$ rows of $M$ are linearly independent.

We show that matrix $B$ is non-singular: let $D$ be the $d \times d$ diagonal matrix, where $D_{i,i} = i^d$; then, $B = D \cdot V^{(d)}$. Matrices $V^{(d)}$ and $D$ are non-singular; thus, $B$ is non-singular as well. Therefore, the first $d$ rows of $M$ are linearly independent.

Assume that some linear combination $c$ of the first $t$ rows of $M$ equals a linear combination of the last $d+1$ rows. Then, the same linear combination $c$, restricted to the columns of $B$, yields a row of $d$ zeros, in contradiction with the non-singularity of matrix $B$.

We remark that the polynomial $p(\cdot)$ can be efficiently computed. Furthermore, given a degree $t \geq d$, it is possible to efficiently sample a random polynomial of degree $t$ that satisfies conditions (1) and (2) of the lemma. □

**Lemma 4.4** *Let $m \geq d+1$, and let $f_1(\cdot) \ldots f_m(\cdot)$ and $g_1(\cdot) \ldots g_m(\cdot)$ be polynomials of degree $d$ over a field $F$ with $|F| \geq m$, such that for every $1 \leq i \leq d+1$ and every $1 \leq j \leq m$ we have $f_i(j) = g_j(i)$ and $g_i(j) = f_j(i)$. Then, there exists a unique polynomial $h(\cdot,\cdot)$ of degree $d$ in two variables so that for every $1 \leq i \leq m$ we have $h(\cdot, i) = f_i(\cdot)$ and $h(i, \cdot) = g_i(\cdot)$.*

**Proof:** Let $E$ be the $(d+1)\times(d+1)$ matrix where $E_{i,j}$ is the coefficient of $x^j$ in $f_i(x)$. Then, the $(i,j)$th entry in $E \cdot V$ is $f_i(j)$.

Let $H \stackrel{\triangle}{=} (V^T)^{-1} \cdot E$, and let $h(x,y)$ be the polynomial of degree $d$ in two variables[23] where the coefficient of $x^i y^j$ is $H_{i,j}$. Then, for every $1 \leq i, j \leq d+1$ we have

$$h(i,j) = V^T \cdot H \cdot V = E \cdot V = f_i(j) = g_j(i).$$

Consequently, for every $1 \leq i \leq d+1$, the polynomials $f_i(\cdot)$ and $h(i, \cdot)$ are two polynomials of degree $d$ that are equal in $d+1$ places. Thus, $f_i(\cdot) = h(i, \cdot)$. Similarly, $g_i(\cdot) = h(\cdot, i)$.

[23] Namely, $h(x,y) = \sum_{i=0}^{d} \sum_{j=0}^{d} a_{i,j} \cdot x^i y^j$, where $a_{0,0}, \ldots, a_{d,d}$ are fixed coefficients.

Now, consider an index $d+1 < i \leq m$. For every $1 \leq j \leq d+1$, we have $f_i(j) = g_j(i)$; by the construction of polynomial $h(\cdot,\cdot)$, we also have $h(\cdot, j) = g_j(i)$. Namely, $f_i(\cdot)$ and $h(i,\cdot)$ are polynomials of degree $d$ that are equal in $d+1$ places. Consequently, $f_i(\cdot) = h(i,\cdot)$. Similarly, $g_i(\cdot) = h(\cdot, i)$. □

**Lemma 4.5** *Let $h(\cdot,\cdot)$, $h'(\cdot,\cdot)$ be two polynomials of degree $d$ in two variables over a field $F$ with $|F| > d$, and let $v_1, \ldots, v_{d+1}$ be distinct elements in $F$. Assume that for every $1 \leq i,j \leq d+1$ we have $h(v_i, v_j) = h'(v_i, v_j)$. Then, $h(\cdot,\cdot) = h'(\cdot,\cdot)$.*

**Proof:** Let $W$ denote the $(d+1)\times(d+1)$ (Vendermonde) matrix defined by $W_{i,j} = (v_j)^i$. Let $H$ be the $(d+1)\times(d+1)$ matrix where $H_{i,j}$ is the coefficient of $x^i y^j$ in $h(x,y)$. Let $H'$ be similarly defined with respect to $h'(x,y)$. Using these notations, we have

$$W^T \cdot H \cdot W = W^T \cdot H' \cdot W.$$

Note that $W$ is non-singular, since $v_1, \ldots, v_{d+1}$ are distinct. Consequently, $H = H'$. □

**Lemma 4.6** *Let $F$ be a field with $|F| > d$, and let $s \in F$. Then, for every sequence $f_1(\cdot), \ldots, f_t(\cdot), g_1(\cdot), \ldots, g_t(\cdot)$ of polynomials of degree $d$, such that $f_i(j) = g_j(i)$ for every $1 \leq i,j \leq d$, there exists a unique polynomial $h(\cdot,\cdot)$ of degree $d$ in two variables with $h(0,0) = s$, so that for every $1 \leq i \leq d$ we have $h(\cdot, i) = f_i(\cdot)$ and $h(i,\cdot) = g_i(\cdot)$.*

**Proof:** Let $E$ be the following $(d+1)\times(d+1)$ matrix. For $1 \leq i,j \leq d$ let $E_{i,j} = f_i(j) = g_j(i)$; let $E_{j,0} = g_j(0)$, and $E_{0,i} = f_i(0)$. Finally, let $E_{0,0} = s$. Let $H \triangleq (V^T)^{-1} \cdot E \cdot V^{-1}$, and let $h(x,y)$ be the polynomial of degree $d$ in two variables, such that the coefficient of $x^i y^j$ is $H_{i,j}$. Similar arguments to those of the proof of Lemma 4.26 show that $h(x,y)$ is the required, uniquely defined, polynomial. □

**Lemma 4.8** *Let $F$ be a finite field with $|F| > d$, and let $A(\cdot)$ and $B(\cdot)$ be polynomials of degree $d$ over $F$. Then, for every sequence $a_{1,1}, \ldots, a_{d,d}, c_1, \ldots, c_d$ of elements in $F$ there exists a unique sequence $\hat{S}_1(\cdot), \ldots, \hat{S}_d(\cdot)$ of polynomials of degree $d$ such that:*

1. *Fore each $1 \leq i,j \leq d$, we have $\hat{S}_i(j) = a_{i,j}$.*
2. *Let $C(\cdot)$ be the truncation to degree $d$ of the polynomial $D(x) = A(x)\cdot B(x) + \sum_{j=1}^{t} x^j \cdot \hat{S}_j(x)$. Then, for $1 \leq i \leq d$ we have $C(i) = c_i$.*

**Proof:** Let $a_{1,1}, \ldots, a_{d,d}, c_1, \ldots, c_d$ be a sequence of field elements, and let $s_{i,j}$ be the coefficient of $x^j$ in the $i$th polynomial in a sequence $\hat{S}_1(\cdot), \ldots, \hat{S}_d(\cdot)$ of polynomials satisfying requirements 1 and 2 of the Lemma. Then, requirements 1 and 2 translate to $d^2+d$ equations in the $d^2+d$ variables $s_{1,0} \ldots s_{d,d}$. It remains to show that the $(d^2+d)\times(d^2+d)$ matrix $M$ of the coefficients of these equations is non-singular. Matrix $M$ has the following form:

$$M = \begin{bmatrix} W_1 & 0 & 0 & \ldots & 0 \\ 0 & W_2 & 0 & \ldots & 0 \\ \ldots & & & & \\ \ldots & & & & \\ 0 & 0 & \ldots & 0 & W_d \\ U_1 & U_1 & \ldots & & U_d \end{bmatrix}$$

where each sub-matrix $W_i$ (of dimensions $d \times (d+1)$) is constructed by deleting the last row from $V^{(d+1)}$, and every matrix $U_i$ (of dimensions $d \times (d+1)$) is constructed by deleting the $i-1$ leftmost columns from a $V^{(d)}$ matrix and appending $i$ zero columns on the right.

Clearly, the first $d^2$ rows in $M$ are linearly independent, and the last $d$ rows in $M$ are linearly independent. It remains to show that no non-trivial linear combination of the first $d^2$ rows equals a linear combination of the last $d$ rows.

Let the $(d^2+d)$-vector $L_1$ be a linear combination of the first $d^2$ rows, let the $(d^2+d)$-vector $L_2$ be a linear combination of the last $d$ rows, and assume that $L_1 = L_2$. We show that $L_1 = L_2 = 0$. Partition $L_2$ to $d$ blocks, where each block contains $d+1$ elements (the $i$th block is a combination of the elements in $U_i$).

The $d$ rightmost columns of $U_d$ are zero, and the $d$ rightmost columns of $W_d$ are linearly independent. Thus, $L_1 = L_2$ involves no rows of $W_d$. Consequently, *all* the last $d+1$ entries (namely, the $d$th block) in $L_2$ are zero.

However, $U_{d-1}$ is a shift-right by one of $U_d$. Thus, the $d$ rightmost entries in the $(d-1)$th block of $L_2$ are zero as well. Using a similar argument to that of the last paragraph, we get that *all* the entries in the $(d-1)$th block in $L_2$ are zero.

Applying this argument $d-2$ more times, we get that $L_1 = L_2 = 0$. □