Feldman's Verifiable Secret Sharing for a Dishonest Majority

Yi-Hsiu Chen* Yehuda Lindell*

January 8, 2024

Abstract

Verifiable secret sharing (VSS) protocols enable parties to share secrets while guaranteeing security (in particular, that all parties hold valid and consistent shares) even if the dealer or some of the participants are malicious. Most work on VSS focuses on the honest majority case, primarily since it enables one to guarantee output delivery (e.g., a corrupted recipient cannot prevent an honest dealer from sharing their value). Feldman's VSS is a well known and popular protocol for this task and relies on the discrete log hardness assumption. In this paper, we present a variant of Feldman's VSS for the dishonest majority setting and formally prove its security. Beyond the basic VSS protocol, we present a publicly-verifiable version, as well as show how to securely add participants to the sharing and how to refresh an existing sharing (all secure in the presence of a dishonest majority). We prove that our protocols are UC secure, for appropriately defined ideal functionalities.

1 Introduction

In this paper, we prove the security of Feldman's verifiable secret-sharing scheme (VSS) [5] in the ideal/real model paradigm of secure multiparty computation. We prove security for a dishonest majority, achieving security *with abort*. More specifically, we consider n parties and a threshold t required to reconstruct the secret. However, we make no limitation on t, and security is guaranteed for any $t \leq n$ (and even $t = n$). Beyond the basic VSS, we consider a number of variants and additional operations that are useful in the threshold cryptography setting. In particular, we construct protocols and prove security for the following:

1. *Basic Feldman VSS:* This is the basic Feldman secret sharing with a dealer and n parties who participate. We stress that we do not assume an honest majority, and as such parties may abort (and some honest parties may abort while others have output). However, as with standard security with abort, it is guaranteed that the output of all honest parties who do not abort is consistent with the same valid sharing.

2. *Feldman VSS with online and offline parties:* In the basic VSS, all n parties send and receive messages. However, consider a setting where t-of-n parties participate in a distributed key generation protocol. In this case, t parties actually participate, and the other $n - t$ parties just receive shares. This is sufficient since anyway any t parties can learn the key, and so it

*Applied Cryptography Group, Coinbase.

suffices for just t parties to generate the key (as we assume that strictly less than t parties are corrupted). Now, in regular Feldman VSS, it is possible to have the $n-t$ parties be passive and merely receive their shares. However, what happens if some of those parties are offline. This can make sense if $n \gg t$, or some of those $n-t$ parties are actually just backup entities. In this case, there is no way to know if a party has received a valid share until they connect. We utilize publicly-verifiable encryption [4] in order to essentially construct a publicly-verifiable secret-sharing scheme [11, 10], where the t parties who are online can all verify that the shared values to the $n-t$ offline parties are valid before the protocol terminates. This ensures that the sharing is valid, even if not all parties are connected. We note that the online parties may abort, but we achieve the guarantee that *if any honest online party completes without aborting, then all honest offline parties are guaranteed to generate output and not abort.* This is very significant in practice where an abort for online parties can be immediately detected and dealt with, but offline parties may discover problems a lot later, and dealing with it is more challenging.

3. *Adding a party:* In some real-world applications where a key is shared among parties, it is necessary to be able to support the addition of new parties (without changing the threshold t). In particular, the set of parties who can approve a signing operation in a threshold signing setting may be dynamic (e.g., consider employees of a specific team at a cryptocurrency custodian). As a result, it is necessary to support adding and removing parties. Our protocol for adding a party works by having t parties subshare the new share to each other, and then send the sum of these subshares – which constitutes a random sharing of the new share – to the new party.

4. *Refresh:* In order to achieve a level of proactive security where it isn't possible to slowly steal shares from one party at a time, we provide a protocol that refreshes an existing secret sharing. The result of a refresh operation is that the parties all hold shares on an independent polynomial that defines the same secret. This is achieved by t parties using VSS to share polynomials that have the secret 0, and then adding all of those shares to the existing one.

5. *Removing a party:* In the same way that it is sometimes necessary to add a party to an existing secret sharing, it is also necessary to revoke a share and remove a party from the sharing. This can be achieved simply by running the refresh operation, without providing the new share to the party being removed. This works since all other parties now hold shares on an independent polynomial, rendering the revoked party's share useless.

We remark that all of our protocols work with more general access structures than just a basic threshold. In particular, we can support any tree with AND, OR and threshold nodes, using standard methods.

Feldman's VSS overview: The idea behind Feldman's VSS is to augment a regular Shamir sharing with a broadcast of the sharing polynomial "in the exponent". That is, let $b_0 = s$ and let $b(x) = \sum_{k=0}^{t-1} b_k \cdot x^k$. Then, in addition to sending each party their share, the dealer broadcasts $B_0, B_1, \ldots, B_{t-1}$ where each $B_k = b_k \cdot G$. Each party P_j, who is supposed to receive the share $s_j = b(\alpha_j)$, then verifies that $s_j \cdot G = \sum_{k=0}^{t-1} (\alpha_j)^k \cdot B_k$, ensuring that all shares are consistent with the single broadcasted polynomial in the exponent.

The main differences between our protocol and the standard Feldman VSS for an honest majority are as follows:

- Since we anyway achieve only security with abort, we do not need a full-blown secure broadcast of the vector $(B_0, \ldots, B_{t-1})$ and it suffices for the parties to run a simple echo-broadcast of what they received. This ensures consistency between all honest parties that do not abort.
- In order to extract the polynomial that is shared in the proof of security when the dealer is corrupted, we need to have the dealer prove knowledge of the polynomial with a zero-knowledge proof of knowledge. In the honest majority setting (with the number of corrupted being less than half of the quorum required, so less than $t/2$ here) this isn't needed since the polynomial can be extracted by the simulator receiving enough shares to reconstruct directly. This is because the number of honest parties in that setting is greater than the degree of the polynomial, something that isn't guaranteed in our setting (where up to $t-1$ parties may be corrupted).
- Standard Feldman VSS has a "complaint" phase where parties can complain that they received an incorrect value and this is fixed by the dealer. In our case with no honest majority, if this occurs then a party will just abort, and so we don't need to "fix" anything. Importantly, as described above, we do ensure that all honest parties receive the same vector $(B_0, B_1, \ldots, B_{t-1})$, and this guarantees that all shares are on the same degree-$(t-1)$ polynomial. In the standard VSS setting, this would require a full-blown secure broadcast (since guaranteed output delivery is needed), whereas in our setting we can use a simple echo-broadcast with just one round where all parties send each other the vector they received. If a party did not receive the same output from everyone, then we just allow it to abort. This ensures that all honest parties who do output something have received the same vector from all other parties. The overall number of rounds is just two (over point to point channels).

On the security of Feldman VSS: Feldman VSS is often considered to not be "fully secure" unless the value being shared is a hard-core bit of the discrete log of the secret. This is because Feldman secret sharing reveals the value $S = s \cdot G$, where s is the secret being shared. This does not meet a standard definition of secret sharing where the secret must remain completely secret, since $s \cdot G$ reveals some information on s. Nevertheless, in the context of elliptic-curve threshold cryptography where the sharing is of an elliptic curve key, the secret s being shared is either a private key or a share of the private key. In such cases, the public key, which is exactly $s \cdot G$ is supposed to be revealed. In addition, when VSS is used for distributed key generation, then each party shares some s_i and the resulting private key is $s = \sum s_i$. In standard distributed key generation protocols, each $s_i \cdot G$ is also revealed. Thus, Feldman VSS can be used without any loss or compromise in security. Indeed, for these applications, revealing $s_i \cdot G$ is exactly what is needed in order to tie the shared values back to the actual key.

Our formalization of security actually bypasses this issue by having the ideal functionality itself provide the exponents of the polynomial $(B_0, \ldots, B_{t-1})$ to all parties. Thus, the security guarantees provided by the functionality are primarily those of correctness and consistency (all honest parties are guaranteed to have valid shares on the same polynomial). In terms of "hiding", this is only suitable for applications where the shares in the exponent can all be revealed. As discussed, this suffices for applications like elliptic-curve threshold cryptography, which is growing in use today.

On the use of Feldman VSS for key generation: In general, VSS protocols reveal nothing about the shared value. As a result, the classic way of running key generation via VSS is for each party to share a secret *in parallel*, and to then sum the result. Of course, in the case that one also needs to obtain the public key, it is necessary to compute that while ensuring that it indeed matches the sum of the shared values. When using Feldman VSS, this latter task is trivial: as described above, each party's sharing reveals its associated "public key share", and the public key is just the sum of these values. However, when using Feldman VSS, if all parties just share their secret in parallel, then it is possible for the adversary to bias the result. This is because they can see the public shares of the honest parties (i.e., $B_0, \ldots, B_{t-1}$) before they send their own shares. As a result, in order to obtain distributed key generation with full simulation, each party first sends an (extractable and equivocal) *commitment* to their VSS sharing. After receiving all commitments, the parties decommit, and simply sum the result. This prevents corrupted parties from biasing the result since they are committed to their sharings before learning anything about the honest parties' sharings.

Rounds of communication and asynchronous computation: In some settings, and in particular in the threshold signing setting, it is possible that some parties involved in the operations (distributed key generation, adding a party, and so on) are humans with their personal devices. In these cases, protocols with many rounds of interaction are problematic, since ensuring that everyone is online together can be challenging. As a result, we aim for a minimal number of rounds. All of our protocols have two rounds, making it sufficient for each party to "connect" twice. That is, a human can connect and participate in the first round, and at a later time connect and participate in the second round, meaning that the connections can be *asynchronous* (except for knowing that the first round has completed). We also support having parties offline and later receiving their output, as described above. In this paper, we call parties asynchronous if they can be online to carry out computations, but we cannot require them to be online at the same time. As a result, they can each connect, download information from some "coordinator machine", prepare a message to be sent to other parties that is sent to the coordinator, and then disconnect.

In the specific case of refresh, we also provide a variant with just a single round. This is due to the fact that distributed key generation, adding a party, and removing a party, are all less common tasks. In contrast, refresh is something that should be run periodically, and here having two rounds can be problematic. Clearly, it is impossible to achieve consensus with only a single round of communication (each party sending something). However, if there are also some "fully online parties" then we show that it's possible to run the consensus between these fully online parties only. The security guarantee is weaker (requiring at least one honest fully online party) but enables us to achieve practical refresh with asynchronous parties.

Security model and composition: We prove security for the stand-alone definition of secure multiparty computation [2, 7] for security with abort (where some honest parties may have output and some may abort) and with no honest majority. In this model, all parties send their inputs to the ideal functionality (computed by a trusted party). The ideal functionality then sends the (ideal-model) adversary the corrupted parties' outputs, and the adversary then instructs the ideal functionality as to which honest parties should receive output. In some cases, the functionality may be interactive, with the adversary interacting with the functionality. This is used to model issues like the fact that the adversary may be able to influence the secret sharing polynomial, but in no way that affects security. For example, it can choose its sharing as a function of shares it receives from the honest parties. This is inconsequential, but must be included in the functionality definition.

Although we prove security in the stand-alone model that guarantees security under sequential composition only, we are really interested in UC security [3]; i.e., security under concurrent general composition. This is achieved by all our protocols, since they are all *perfectly secure* with *straight-line simulation* (i.e., no rewinding). As shown in [8], this implies UC security. We remark that the actual VSS protocol is only perfectly secure in the ideal zero-knowledge hybrid model. However, this suffices since it means that as long as the zero-knowledge proof of knowledge is instantiated with a UC-secure protocol, then everything is UC secure.

A note on novelty: To the best of our knowledge, a formal description and proof of security for Feldman's VSS in the case of a *dishonest majority* has not appeared previously in the literature. Our protocols are based on well-known techniques, but have not previously been formalized and proven. This has value, for example, to ensure that zero-knowledge proofs are used in the sharing (something not always done in naive implementations). In addition, our simple protocols for adding a party and refreshing a sharing have also, to the best of our knowledge, not appeared previously.

2 Definitions and Preliminaries

2.1 Preliminaries

Lagrange interpolation: Let $\alpha_1, \ldots, \alpha_n$ be distinct field elements. We denote the Lagrange basis polynomials with respect to a set $\mathcal{I} \subseteq [n]$ by $\left\{L_i^{\mathcal{I}}\right\}_{i\in\mathcal{I}}$ where $L_i^{\mathcal{I}}(x) = \prod_{j\in\mathcal{I}\setminus\{i\}} \frac{x-\alpha_j}{\alpha_i-\alpha_j}$. The standard Lagrange interpolation works by the fact that for any set of t distinct points $\{(\alpha_i, \beta_i)\}_{i\in\mathcal{I}}$, it holds that $f(x) = \sum_{i\in\mathcal{I}} \beta_i \cdot L_i^{\mathcal{I}}(x)$ is the unique degree-$(t-1)$ polynomial such that $f(\alpha_i) = \beta_i$ for every $i \in \mathcal{I}$.

Zero-knowledge: We describe and prove our protocols secure with an ideal functionality for a (batch) zero-knowledge proof of knowledge of the discrete log of a series of group elements. The relation is formally defined by:

$$\mathsf{BatchDL} = \left\{ \{X_i\}_{i\in[k]}, \{x_i\}_{i\in[k]} : \forall i \in [k] \ x_i \cdot G = X_i \right\}.$$

We denote an ideal zero-knowledge proof of knowledge functionality for this functionality by $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$. This can be realized with UC security in the random-oracle model by applying the Fischlin transform [6] to the standard Sigma protocol for discrete log by Schnorr [9], in parallel for each value. The functionality is a pairwise functionality, and so the prover P_i sends $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{X_i\}_{i\in[k]}\right)$, and $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ sends $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{X_i\}_{i\in[k]}\right)$ to party P_j if the proof is valid, and sends $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{X_i\}_{i\in[k]}, \mathsf{abort}\right)$ if it is invalid. Practically, we realize this by having each P_i generate a non-interactive proof and simply send it to each party separately. We stress that this means that the functionality does not guarantee that all parties receive the same proof. In our protocols this doesn't matter, since we anyway guarantee that the statement being proven is the same for all parties, and this suffices.

Publicly-verifiable encryption: In the context of our applications here, publicly-verifiable encryption is an encryption scheme with an additional property that it is possible to verify (in zero knowledge) that the encrypted value is the discrete log of a given point. That is, such a scheme has standard encrypt and decrypt functionality, denoted $c = \mathsf{vencrypt}_{pk}(x; X)$ and $x = \mathsf{vdecrypt}_{sk}(c)$, along with a verification function $\mathsf{enc\text{-}verify}_{pk}(c, X)$ that outputs 1 if and only if c is an encryption of the discrete log of X under public key pk. A simple and canonical construction of such a scheme (that works with any encryption scheme) can be found in [4]. The following ideal functionality, denoted $\mathcal{F}_{\mathsf{PVE}}$, models publicly-verifiable encryption and runs with parties $P_1, \ldots, P_n$:

- **Init:** after receiving init from all parties $P_1, \ldots, P_n$ (for setting up a PKI), proceed. (Before receiving all init messages, ignore all other messages.)
- **Encrypt:** upon receiving $(\mathsf{vencrypt}, i, j, m)$ from a party P_i with $m \in \mathbb{Z}_q$
  1. Compute $M = m \cdot G$
  2. Store (i, j, m, M)
  3. Send $(\mathsf{vencrypted}, i, j, M)$ to all parties $P_1, \ldots, P_n$
- **Decrypt:** upon receiving $(\mathsf{vdecrypt}, i, j, m, M)$ from party P_j
  1. Ignore unless (i, j, m, M) has been stored
  2. Send $(\mathsf{vdecrypted}, i, m)$ to party P_j

2.2 Ideal Functionality Definitions

In this section we define the ideal functionalities for each of the operations we support. We do not define a "remove party" functionality since, as we have described, this is easily carried out by just running refresh without including the party to be removed.

Honest parties' outputs: As described above, we consider a setting of security with abort, where some honest parties may abort while others receive output. This is modeled by having the adversary send the trusted party/ideal functionality the set of honest parties to receive output. This can be modeled within the instructions of the adversary, or as part of the ideal-model execution. We choose the latter, with the understanding that these are equivalent. We denote the list of honest parties sent by the ideal adversary/simulator in the ideal execution that should receive output by $\mathcal{O}_h$.

2.2.1 VSS

We define a VSS functionality $\mathcal{F}_{\mathsf{vss}}$ for sharing a secret s via a degree-$(t-1)$ polynomial $s(x)$. The functionality also sends all of the polynomial coefficients "in the exponent" (i.e., $b_k \cdot G$ for $k = 0, \ldots, t-1$ where $s(x) = \sum_{k=0}^{t-1} b_k \cdot x^k$)). These additional values help to enforce correct behavior (in the dishonest majority setting). Recall that, as we have discussed, releasing $S = s \cdot G$ is not "leakage" per se in our application, since we use this functionality for the case that s is an EC private key (or a share of an EC private key). In this case, S is its associated public key which is supposed to be public.

Functionality $\mathcal{F}_{\mathsf{vss}}$:

- Upon receiving input from party P_i with $i \in [n]$
  - $(\mathsf{share}, \mathsf{sid}, s)$ if P_i is honest
  - $(\mathsf{share}, \mathsf{sid}, s(x))$ if P_i is corrupted

  operate as follows:

  1. If P_i is honest, then choose a random degree-$(t-1)$ polynomial $s(x)$ with $s(0) = s$
  2. If P_i is corrupted, then ignore the message if $\deg(s(x)) \geq t$
  3. Let $s(x) = \sum_{k=0}^{t-1} b_k \cdot x^k$[1]
  4. For $k = 1, \ldots, t-1$, compute $B_k = b_k \cdot G$
  5. Set $\mathcal{B} = (B_0, \ldots, B_{t-1})$
  6. For $j = 1, \ldots, n$, compute $s_j = s(\alpha_j)$
  7. Send $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$ to party P_j for $j = 1, \ldots, n$

A more minimal functionality? The $\mathcal{F}_{\mathsf{vss}}$ functionality reveals to the parties not only $S = B_0 = s \cdot G$, but also *all* of the polynomial coefficients in the exponent $B_1, \ldots, B_{t-1}$ for all parties. In our uses of $\mathcal{F}_{\mathsf{vss}}$, the value $S = B_0 = s \cdot G$ is always revealed, as it is the public key and s is the private key (or they are shares of the public/private keys but also revealed). However, the additional $B_1, \ldots, B_{t-1}$ need not be revealed in principle. As such, it may seem that this formulation reveals more information than necessary, and it would be better to have the functionality send only $(\mathsf{share}, \mathsf{sid}, S, s_j)$ where $S = s \cdot G$ for the secret s. However, it is easy to see that the basic Feldman secret sharing does *not* securely compute a minimal functionality where parties only receive $(\mathsf{share}, \mathsf{sid}, S, s_j)$ from the ideal functionality. This is because the distinguisher sees all outputs – of both honest and corrupted parties. Now, in the real execution, the corrupted parties see the polynomial "in the exponent" and so can compute $S_1, \ldots, S_n$ where $S_j = s_j \cdot G$ for all parties, including the honest parties. In contrast, in an ideal execution where less than t parties are corrupted, the simulator *cannot* compute $s_j \cdot G$ for an honest P_j, given only the points of the corrupted parties. Thus, this more minimal functionality cannot be computed in this way, and any application using Feldman VSS will have to be proven secure for the above functionality, where $B_0, \ldots, B_{t-1}$ (or equivalently, $S_0, S_1, \ldots, S_n$) are all revealed. Fortunately, for applications like distributed key generation and the like, this additional information can be simulated by choosing a random polynomial "in the exponent", and so it is inconsequential.

Observe also that a corrupted party sends $s(x)$ and not just s (like an honest dealer) since in the real protocol, nothing forces a corrupted dealer to use a random polynomial. It is possible to securely realize a stronger functionality where a corrupted dealer sends s to $\mathcal{F}_{\mathsf{vss}}$ and the functionality chooses $s(x)$, like for an honest dealer. Realizing this functionality would require a type of coin tossing where the dealer commits to a sharing of s and all other parties commit to a sharing of 0, and then all parties decommit and the sharing is the sum of all sharings. However, this is not needed in applications of secret sharing we are familiar with, and so would add unnecessarily complexity and cost.

[1] We are aware that denoting the polynomial by $s(x)$ and the coefficients by b_k and not s_k looks strange. However, we do this since we denote the i'th party's share by s_i.

Public verifiability: Publicly-verifiable secret sharing enables anyone to verify that each party's (encrypted) share is valid. This can be achieved canonically (with Feldman's VSS) by simply having the dealer encrypt each party's share under the recipient's public key using publicly-verifiable encryption. Then, given the coefficients of the sharing polynomial in the exponent, it is possible for anyone to compute any party's share in the exponent, and then verify its validity in the encryption. We do not model public verifiability any differently in the definition. Rather, we use it as a way of computing the standard $\mathcal{F}_{\mathsf{vss}}$ with only a quorum t of actively participating parties. In particular, we do not need the $n-t$ additional parties to be online during the secret sharing itself. This is achieved by having the t online parties verify that the encrypted shares of all parties are valid, without the $n-t$ additional parties needing to be online. We remark that public verifiability *does not guarantee* output delivery. However, it does provide us with a weaker version of guaranteed output delivery that is very meaningful. In particular, we achieve the property that *if any honest online party does not abort* then it is guaranteed that *all offline honest parties do not abort.*

More general access structure: Our protocols all support access structures of a more general form of any tree of AND, OR and threshold nodes. The extension of the sharing to access structures of these types is straightforward. We therefore focus on the basic threshold case only; the proof remains essentially the same for the general case.

2.2.2 Add Party

We also define a functionality $\mathcal{F}_{\mathsf{add}}$ for adding a new party to the sharing. This involves computing $s(\alpha_{n+1})$ for a new P_{n+1}.

Functionality $\mathcal{F}_{\mathsf{add}}$: Upon receiving $(\mathsf{add}, \mathsf{sid}, \mathcal{B}, s_i, \alpha_{n+1})$ from t parties $\mathcal{I} \subseteq [n]$,

1. Verify that all received $\mathcal{B}$ and α_{n+1} are the same
2. Parse $\mathcal{B} = (B_0, \ldots, B_{t-1})$
3. Verify that $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ for all $i \in \mathcal{I}$
4. Reconstruct $s(x)$ to be the unique degree-$(t-1)$ polynomial such that $s(\alpha_i) = s_i$ for all $i \in \mathcal{I}$
5. Compute $s_{n+1} = s(\alpha_{n+1})$
6. Send $(\mathsf{add}, \mathsf{sid}, \mathcal{B})$ to the adversary
7. Send $(\mathsf{add}, \mathsf{sid}, \mathcal{B}, s_{n+1})$ to P_{n+1}

We remark that in the protocol computing this, P_{n+1} is a passive recipient only.

2.2.3 Refresh

Finally, we define a functionality $\mathcal{F}_{\mathsf{refresh}}$ for refreshing the sharing. This involves interactively computing a sharing of a new random polynomial s' for the sharing, with the constraint that $s'(0) = s(0)$. As we will see below, this works by having all parties choose a random sharing of 0, and then defining the refreshed sharing to be the sum of the original sharing plus all the new ones. This is clearly a new sharing of the same value. A naive definition of this functionality would be for

$\mathcal{F}_{\mathsf{refresh}}$ to simply reconstruct the secret and choose a new random polynomial with the same secret. However, securely realizing this functionality would be difficult since it would require the result to be a completely random polynomial. As such, all parties would have to commit to their sharings of 0 and then decommit (to ensure that no sharing is chosen as a function of the other sharings). However, there is no real need for the polynomial to be truly random. Therefore, we enable the ideal model adversary to receive the new sharing chosen by $\mathcal{F}_{\mathsf{refresh}}$ and to then send a sharing of 0 to be added to it. This models the adversary's ability to bias the sharing polynomial in the real protocol by first seeing the shares of the honest parties and only then sending its sharing. This does not negatively impact security since adding a known polynomial to a secret random polynomial does not leak any information about the secret.

Functionality $\mathcal{F}_{\mathsf{refresh}}$:

- Upon receiving $(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}, s_i)$ from t parties $\mathcal{I}$:
  1. Verify that all received $\mathcal{B}$ are the same
  2. Parse $\mathcal{B} = (B_0, \ldots, B_{t-1})$
  3. Verify that $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ for all $i \in \mathcal{I}$
  4. Reconstruct $s(x)$ to be the unique polynomial such that $s(\alpha_i) = s_i$ for all $i \in \mathcal{I}$
  5. Choose a new random degree-$(t-1)$ polynomial $\hat{s}(x) = \sum_{k=0}^{t-1} \hat{b}_k \cdot x^k$ under the constraint that $\hat{s}(0) = s(0)$
  6. For $k = 1, \ldots, t-1$, compute $\hat{B}_k = \hat{b}_k \cdot G$
  7. Set $\hat{\mathcal{B}} = (\hat{B}_0, \ldots, \hat{B}_{t-1})$
  8. Compute $\hat{s}_j = \hat{s}(\alpha_j)$ for all $j = 1, \ldots, n$
  9. Send $\left(\mathsf{refresh}, \mathsf{sid}, \{\hat{s}(\alpha_i)\}_{i \in \mathcal{I}_c}, \hat{\mathcal{B}}\right)$ to the adversary, where $\mathcal{I}_c$ denotes the set of corrupted parties, and await the adversary's response
- Upon receiving $(\mathsf{refresh}, \mathsf{sid}, \tilde{s}(x))$ from the adversary after sending it $\left(\mathsf{refresh}, \mathsf{sid}, \{\hat{s}(\alpha_i)\}_{i \in \mathcal{I}_c}, \hat{\mathcal{B}}\right)$ (if before that, then ignore):
  1. Verify that $\tilde{s}(x)$ is a degree-$(t-1)$ polynomial and that $\tilde{s}(0) = 0$
  2. Set $s'(x) = \hat{s}(x) + \tilde{s}(x) = \sum_{k=0}^{t-1} b'_k \cdot x^k$
  3. For $k = 0, \ldots, t-1$, compute $B'_k = b'_k \cdot G$
  4. Set $\mathcal{B}' = (B'_0, \ldots, B'_{t-1})$
  5. Compute $s'_j = s'(\alpha_j)$ for $j = 1, \ldots, n$
  6. Send $\left(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}', s'_j\right)$ to party P_j for $j = 1, \ldots, n$
  7. Send $(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}')$ to the adversary

We remark that in the protocol computing this, all parties P_j with $j \notin \mathcal{I}$ are passive recipients, receiving messages only.

3 Protocols

3.1 Securely Computing $\mathcal{F}_{\mathsf{vss}}$ with n Online Parties

Intuition: The protocol works by the dealer computing Shamir shares of the secret, and then sending each party its share as well as the coefficients of the polynomial "in the exponent" (in elliptic-curve notation, this means that each coefficient b_k is given as $B_k = b_k \cdot G$). Since groups support multiplication by a scalar and addition, it is possible for any party to compute the polynomial in the exponent (meaning compute $f(a) \cdot G$ for any a) given a and the coefficients in the exponent (even without knowing f itself). Thus, each party verifies that its share is consistent with the polynomial in the exponent, as well as running an echo-broadcast on the coefficients to ensure that they all received the same polynomial. This ensures that all honest parties hold shares on the same degree-$(t-1)$ polynomial, as required.

Protocol 3.1 (Feldman VSS – Π_{vss})

Parties: *$P_1, \ldots, P_n$ with P_i being the dealer*

Common input: $\mathsf{sid} \in \{0,1\}^*$*, parameters $t, n \in \mathbb{N}$ with $t \leq n$, and unique non-zero values $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$*

P_i's private input: $s \in \mathbb{F}_q$

The protocol:

1. ***Round 1 – party P_i:***
   (a) *Set $b_0 = s$*
   (b) *Choose random $b_1, \ldots, b_{t-1} \in \mathbb{F}_q$ and define $s(x) \stackrel{\text{def}}{=} \sum_{k=0}^{t-1} b_k \cdot x^k$*
   (c) *For every $j \in [n]$, set $s_j = s(\alpha_j)$*
   (d) *For $k = 0, \ldots, t-1$, compute $B_k = b_k \cdot G$*
   (e) *Define $\mathcal{B} = (B_0, \ldots, B_{t-1})$*
   (f) *Send $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{B_k\}_{k=0}^{t-1}, \{b_k\}_{k=0}^{t-1}\right)$ to $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ for every $j \in [n]$*
   (g) *Send $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$ to P_j for every $j \in [n]$*

2. ***Round 2 – each P_j with $j \neq i$:*** *Upon receiving $(\mathit{share}, \mathsf{sid}, \mathcal{B}, s_j)$ from P_i and $(\mathsf{prove}, \mathsf{sid}, \ldots)$ from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$*
   (a) *Parse $\mathcal{B} = (B_0, \ldots, B_{t-1})$*
   (b) *Verify that all B_k are valid group elements (they are allowed to be the identity)*
   (c) *If $s_j \cdot G \neq \sum_{k=0}^{t-1} (\alpha_j)^k \cdot B_k$, then send abort to all parties and abort*
   (d) *If the message from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ is $\left(\mathsf{prove}, \mathsf{sid}, \{B_k\}_{k=0}^{t-1}, \mathsf{abort}\right)$, then send abort to all parties and abort*
   (e) *If the set $\{B_k\}_{k=0}^{t-1}$ from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ is not the same as $\mathcal{B}$, then send abort to all parties and abort*
   (f) *Send $\mathcal{B}$ to all P_ℓ with $\ell \in [n]$*
   (In practice, it suffices to send $H(\mathcal{B})$, where H is a collision-resistant hash function.)

3. ***Output* – *each* P_j*:*** *upon receiving $\mathcal{B}_{1\to j},\ldots,\mathcal{B}_{n\to j}$ (where $\mathcal{B}_{\ell\to j}$ denotes the set $\mathcal{B}$ that P_j received from party P_ℓ)*
   (a) *Abort unless $\mathcal{B}_{1\to j}=\cdots=\mathcal{B}_{n\to j}=\mathcal{B}$*
   (b) *Output* $(\mathsf{share},\mathsf{sid},\mathcal{B},s_j)$

Security: We now prove the security of the protocol for up to $t-1$ corrupted parties. Since t is the minimum quorum size (and t can even equal n), this is the setting of a dishonest majority (i.e., security as long as at least one party is honest).

Theorem 3.2 *Protocol Π_{vss} realizes the functionality $\mathcal{F}_{\mathsf{vss}}$ in the $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$-hybrid model with perfect security-with-abort, in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t\le n$.*

Proof: Let $\mathcal{I}=[n]$ denote the set of all parties, let $\mathcal{I}_c\subseteq\mathcal{I}$ denote the set of corrupted parties, and let $\mathcal{I}_h\overset{\text{def}}{=}\mathcal{I}\setminus\mathcal{I}_c$ denote the set of honest parties. If $\mathcal{I}_c$ is empty, then simulation is trivial; we therefore assume that there is at least one corrupted party. We separately consider the case that the dealer P_i is honest, and the case that the dealer P_i is corrupted.

Case 1 – the dealer P_i is corrupted: The simulator for this case works simply by running the honest parties (this is easy since they have no secret input) and seeing which would abort or not, and by extracting the polynomial from the adversary via the zero-knowledge proofs of knowledge to send to the ideal functionality $\mathcal{F}_{\mathsf{vss}}$. Let $\mathcal{A}$ be the real-world adversary. We construct an ideal world adversary/simulator $\mathcal{S}$, as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ with sid and receives the messages $\left(\mathsf{prove},i,j,\mathsf{sid},\{B_k\}_{k=0}^{t-1},\{b_k\}_{k=0}^{t-1}\right)$ intended for $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ and $(\mathsf{share},\mathsf{sid},\mathcal{B}_j,s_j)$, for all $j\in\mathcal{I}_h$.
2. For every $j\in\mathcal{I}_h$:
   (a) $\mathcal{S}$ verifies that $s_j\cdot G=\sum_{k=0}^{t-1}(\alpha_j)^k\cdot B_k$ and that $B_k=b_k\cdot G$ for every $k=0,\ldots,t-1$ in the prove message for P_j
   (b) If yes (to both), $\mathcal{S}$ simulates P_j sending $\mathcal{B}_j$ to all parties
   (c) Else, $\mathcal{S}$ simulates P_j sending abort to all parties, and sends abort to $\mathcal{F}_{\mathsf{vss}}$ (and after simulating all honest parties sending their message in this round, $\mathcal{S}$ simulates all honest parties aborting)
3. $\mathcal{S}$ receives the message $\left\{\mathcal{B}_j^\ell\right\}_{j\in\mathcal{I}_c}$ from $\mathcal{A}$ for every $\ell\in\mathcal{I}_h$, where $\mathcal{B}_j^\ell$ is the vector sent from the corrupted party P_j to honest party P_ℓ (note that P_j may send different vectors $\mathcal{B}_j$ to different honest parties)
4. For every $\ell\in\mathcal{I}_h$
   (a) $\mathcal{S}$ verifies that the $\mathcal{B}$ vectors received by P_ℓ are all the same (based on the $\mathcal{B}_j$ values computed for the honest parties, and the $\mathcal{B}_j^\ell$ values received for P_ℓ)
   (b) If yes, $\mathcal{S}$ adds ℓ to $\mathcal{O}_h$ (the set of honest party to receive output, initially empty)

5. $\mathcal{S}$ defines $s(x) = \sum_{k=0}^{t-1} b_k \cdot x^k$ and sends $(\mathsf{share}, \mathsf{sid}, s(x))$ to $\mathcal{F}_{\mathsf{vss}}$, together with the list of honest parties $\mathcal{O}_h$ to receive output

6. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs

If $\mathcal{A}$ sends an incorrect proof to $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ to an honest party, or sends an incorrect s_j to an honest party, then in the real execution all honest parties abort (since they instruct all honest parties to abort). Likewise, in the ideal execution, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{vss}}$ and so all honest parties abort in the ideal execution. Next, if $\mathcal{A}$ sent a share message with different sets $\mathcal{B}$ and $\mathcal{B}'$ to two different honest parties, then all honest parties abort in the output phase, in both the real and ideal executions. This holds since all honest parties receive different $\mathcal{B}$ and $\mathcal{B}'$ in the second round. If none of the above happens, then it is guaranteed that all honest parties received the same set $\mathcal{B}$, and they all received a valid share s_j such that $s_j \cdot G = \sum_{k=0}^{t-1} (\alpha_j)^k \cdot B_k$. Thus, any honest party not aborting will output $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$ with the same set $\mathcal{S}$, as in the ideal execution (since $\mathcal{F}_{\mathsf{vss}}$ computes $\mathcal{B}$ in the same way based on $s(x)$). Finally, note that if any corrupt party sends a different $\mathcal{B}'$ value to an honest party in the last step, then that party will abort in both the real and ideal executions. Thus, the distribution over the adversary and honest party's outputs are identical in both cases.

Case 2 – the dealer P_i is honest: The simulation in this case works by simulating the messages that the corrupted parties would receive, using the shares received from $\mathcal{F}_{\mathsf{vss}}$. Let $\mathcal{A}$ be the real-world adversary. We construct an ideal world adversary/simulator $\mathcal{S}$, as follows:

1. $\mathcal{S}$ receives $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$ from $\mathcal{F}_{\mathsf{vss}}$, for every $j \in \mathcal{I}_c$, where $\mathcal{B} = (B_0, \ldots, B_{t-1})$

2. $\mathcal{S}$ invokes $\mathcal{A}$ with sid and simulates $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ sending it $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{B_k\}_{k=0}^{t-1}\right)$ for every $j \in \mathcal{I}_c$, and the honest dealer P_i sending it $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$ for every $j \in \mathcal{I}_c$

3. $\mathcal{S}$ simulates all honest parties sending $\mathcal{B}$ to all corrupted parties in round 2

4. $\mathcal{S}$ receives the messages $\left\{\mathcal{B}_j^\ell\right\}_{\ell \in \mathcal{I}_h}$ that $\mathcal{A}$ sends for every $j \in \mathcal{I}_c$ to all honest parties $\ell \in \mathcal{I}_h$

5. For every $\ell \in \mathcal{I}_h$, if $\mathcal{A}$ sends $\mathcal{B}$ as received to the honest party P_ℓ from every corrupted P_j, then $\mathcal{S}$ adds ℓ to $\mathcal{O}_h$ (the set of honest party to receive output, initially empty)

6. $\mathcal{S}$ sends $\mathcal{O}_h$ to $\mathcal{F}_{\mathsf{vss}}$ to indicate which honest parties receive output

7. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs

In order to see that the distribution over the messages in the real and ideal executions is identical, observe that the honest dealer in the real protocol chooses the polynomial $s(x)$ identically to the way that the functionality $\mathcal{F}_{\mathsf{vss}}$ chooses the polynomial (given only s) in the ideal execution. Furthermore, the only impact that the corrupted parties can have in this execution is to send incorrect $\mathcal{B}$ values. These are easily simulated perfectly by $\mathcal{S}$, as they only impact who receives output and who aborts. This completes the proof. $\square$

Observe that the simulator $\mathcal{S}$ in the proof of Theorem 3.2 is straight line (it does not rewind $\mathcal{A}$). Since perfect security with a straight-line simulator implies UC security, as proven in [8], we have the following corollary:

Corollary 3.3 *Protocol Π_{vss} UC realizes with abort the functionality $\mathcal{F}_{\mathsf{vss}}$ in the $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$-hybrid model, in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t \leq n$.*

3.2 Securely Computing $\mathcal{F}_{\mathsf{vss}}$ with t Online Parties and $n-t$ Offline Parties

Intuition and security goal: Our above protocol considers a scenario where all n parties are online and interacting. However, given that we only obtain security for up to $t-1$ corrupted parties (since we want to be able to reconstruct with any t parties), it actually suffices to have only t parties interact in an online manner in the VSS, and to have the remaining $n-t$ parties be passive and merely receive output. This can be achieved naively by simply having the dealer send its round 1 message to only the online parties, and then having all online parties send their round 2 messages (and the dealer send its round 1 messages) to the offline parties as well (encrypted under each offline party's public encryption key). Upon receiving all of the messages from online parties, the offline parties can just verify that everything is consistent and output their share if yes.

The above simple extension has a disadvantage in practical settings. In particular, offline parties in practice may take a long time until they come online. The desired property would be that whenever they come online, they should be able to receive their share and join any needed computation. However, if the dealer or one of the online parties sends an offline party an incorrect value, then the offline party would abort. Given that this may happen a long time after the shares were generated, dealing with this at that time is costly and painful. As such, we would like to *ensure* that the offline parties can successfully obtain their shares later. This is of course a problem since in the setting of a dishonest majority, it is impossible to achieve guaranteed output delivery. In particular, in our VSS sharing protocol above, it suffices for a corrupted party to send a different $\mathcal{B}'$ vector in round 2 and all honest parties will abort. This is in some sense unavoidable. However, it is possible to achieve a guarantee that if any honest online party accepts then so will all honest offline parties later on. This can be achieved by having all parties sign on the public $\mathcal{B}$, and then each online party sends each offline party *all* of the signatures. If there are t valid signatures on some $\mathcal{B}$ then the offline party accepts that as the value. This prevents a corrupted party sending an invalid $\mathcal{B}'$ in round 2. However, this still doesn't solve the problem that the dealer may send an offline party an incorrect share s_j. This can be prevented by having the dealer send a *publicly-verifiable encryption* of the share for each offline party. Such an encryption has the property that it's possible to efficiently verify that some ciphertext is a valid encryption of the discrete log of some group element. This enables all online parties to verify all offline party shares, and to sign on these ciphertexts together with $\mathcal{B}$. The protocol is described formally below.

Protocol 3.4 ($\Pi^{\mathrm{off}}_{\mathsf{vss}}$)

Parties: *$P_1,\ldots,P_n$ with P_i being the dealer, with $\mathcal{I}_{\mathrm{on}} \subseteq [n]$ the set of t online parties, and $\mathcal{I}_{\mathrm{off}} = [n]\setminus\mathcal{I}_{\mathrm{on}}$ the set of $n-t$ offline parties; note that $i \in \mathcal{I}_{\mathrm{on}}$*

Common input: $\mathsf{sid} \in \{0,1\}^*$*, parameters $t,n \in \mathbb{N}$ with $t \le n$, a vector of public keys $\mathcal{PKI} = (pk_1,\ldots,pk_n)$ and unique non-zero values $\alpha_1,\ldots,\alpha_n \in \mathbb{F}_q$*

P_i's private input: $s \in \mathbb{F}_q$

Each P_j's private input: *a private key sk_j (associated with pk_j in $\mathcal{PKI}$)*

The protocol:

1. ***Round 1 – party P_i:***

   (a) *Set $b_0 = s$*

   (b) *Choose random $b_1,\ldots,b_{t-1} \in \mathbb{F}_q$ and define $s(x) \overset{\mathrm{def}}{=} \sum_{k=0}^{t-1} b_k \cdot x^k$*

   (c) *For every $j \in [n]$, set $s_j = s(\alpha_j)$*
   (d) *For $k = 0, \ldots, t-1$, compute $B_k = b_k \cdot G$*
   (e) *Define $\mathcal{B} = (B_0, \ldots, B_{t-1})$*
   (f) *Send $\left(\mathsf{prove}, \mathsf{sid}, i, j, \{B_k\}_{k=0}^{t-1}, \{b_k\}_{k=0}^{t-1}\right)$ to $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ for every $j \in \mathcal{I}_{\rm on}$*
   (g) *For every $\ell \in \mathcal{I}_{\rm off}$, compute the publicly-verifiable encryption $V_\ell = \mathsf{vencrypt}_{pk_\ell}(s_\ell; S_\ell)$ for $S_\ell = s_\ell \cdot G$*
   (h) *Set $\mathcal{V} = \{V_\ell\}_{\ell \in \mathcal{I}_{\rm off}}$*
   (i) *Compute $\sigma_i = \mathsf{Sign}_{sk_i}(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V})$*
   (j) *Send $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_i, s_j)$ to P_j for every $j \in \mathcal{I}_{\rm on}$*

2. ***Round 2 – each P_j with $j \in \mathcal{I}_{\rm on} \setminus \{i\}$:*** *Upon receiving $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_i, s_j)$ from P_i and $(\mathsf{prove}, \mathsf{sid}, \ldots)$ from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$*
   (a) *Parse $\mathcal{B} = (B_0, \ldots, B_{t-1})$ and $\mathcal{V} = \{V_\ell\}_{\ell \in \mathcal{I}_{\rm off}}$*
   (b) *Verify that all B_k are valid group elements (they are allowed to be the identity) and that all V_ℓ are valid ciphertexts*
   (c) *If $s_j \cdot G \neq \sum_{k=0}^{t-1} (\alpha_j)^k \cdot B_k$, then send abort to all parties and abort*
   (d) *If the message from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ is $\left(\mathsf{prove}, \mathsf{sid}, \{B_k\}_{k=0}^{t-1}, \mathsf{abort}\right)$, then send abort to all parties and abort*
   (e) *If the set $\{B_k\}_{k=0}^{t-1}$ from $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}$ is not the same as $\mathcal{B}$, then send abort to all parties and abort*
   (f) *If there exists $V_\ell \in \mathcal{V}$ such that $\mathsf{enc\text{-}verify}_{pk_\ell}(V_\ell, S_\ell) = 0$ where $S_\ell = \sum_{k=0}^{t-1} (\alpha_\ell)^k \cdot B_k$, then send abort to all parties and abort*
   (g) *Compute $\sigma_j = \mathsf{Sign}_{sk_j}(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V})$*
   (h) *Send $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_j)$ to all P_ℓ with $\ell \in \mathcal{I}_{\rm on}$*

3. ***Round 3 and online party output – each P_j with $j \in \mathcal{I}_{\rm on}$:*** *Upon receiving $(\mathsf{share}, \mathsf{sid}, \mathcal{B}_\ell, \mathcal{V}_\ell, \sigma_\ell)$ from P_ℓ for all $\ell \in \mathcal{I}_{\rm on}$*
   (a) *Each P_j verifies that $\mathsf{Verify}_{pk_\ell}(\mathsf{share}, \mathsf{sid}, \mathcal{B}_\ell, \mathcal{V}_\ell, \sigma_\ell) = 1$ and that $\mathcal{B}_\ell = \mathcal{B}$ and $\mathcal{V}_\ell = \mathcal{V}$ for all $\ell \in \mathcal{I}_{\rm on} \setminus \{i\}$, where $(\mathcal{B}_\ell, \mathcal{V}_\ell)$ are the values received from P_ℓ and $(\mathcal{B}, \mathcal{V})$ are the values received from P_i. If no, it aborts.*
   (b) *Each P_j sends $\left(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \{\sigma_k\}_{k \in \mathcal{I}_{\rm on}}\right)$ to each offline P_ℓ*
   (c) *Each P_j outputs $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$*

4. ***Output for offline parties – each P_j with $j \in \mathcal{I}_{\rm off}$:*** *Upon receiving $\left(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \{\sigma_\ell\}_{\ell \in \mathcal{I}}\right)$ from one or more parties,*
   (a) *If there exists a message $\left(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \{\sigma_\ell\}_{\ell \in \mathcal{I}}\right)$ such that $\mathsf{Verify}_{pk_\ell}(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_\ell) = 1$ for all $\ell \in \mathcal{I}$ and $\mathcal{I}$ is of size t, then compute $s_j = \mathsf{vdecrypt}_{sk_j}(V_j)$ for $V_j \in \mathcal{V}$, and output $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, s_j)$*

Security: The proof of security with abort is almost identical to that of Theorem 3.2. The only difference is that some messages are sent as publicly-verifiable encryptions, but these can be simulated in the same way. Regarding the property that if at least honest online party accepts then so do all honest offline parties, this follows immediately from the fact that the message $\left(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \{\sigma_k\}_{k \in \mathcal{I}_{\text{on}}}\right)$ sent to all offline parties by any honest online party is valid, and will result in the offline party accepting and outputting its decryption. Now, since this message has t valid signatures, it follows that *all honest online parties* viewed the same $\mathcal{B}$ and $\mathcal{V}$, and thus all outputs of online and offline honest parties are consistent, as required. Note that there cannot be more than one $(\mathcal{B}, \mathcal{V})$ with this property, except with negligible probability, since at most $t-1$ parties are corrupted and all honest parties sign on the same $(\mathcal{B}, \mathcal{V})$ that they agree upon in the echo-broadcast consensus. Thus, there can be more than one valid pair $(\mathcal{B}, \mathcal{V})$ only if the adversary can forge honest parties' signatures. The proof of the following theorem is essentially the same as Theorem 3.2, with the addition regarding guaranteed output following the above discussion ($\mathcal{F}_{\mathsf{PVE}}$ denotes the publicly-verifiable encryption ideal functionality).

Theorem 3.5 *Protocol $\Pi_{\mathsf{vss}}^{\mathsf{off}}$ realizes the functionality $\mathcal{F}_{\mathsf{vss}}$ in the $\mathcal{F}_{\mathsf{zk}}^{\mathsf{BatchDL}}, \mathcal{F}_{\mathsf{PVE}}$-hybrid model with perfect security-with-abort, in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t \leq n$. Furthermore, if at least one honest online party accepts then so do all honest offline parties.*

A setting with a coordinator machine instead of point-to-point channels: In a setting where the parties do not have direct point-to-point channels, but rather communicate by sending *encrypted-and-signed messages* via some central "coordinator machine", there are a few changes that need to be highlighted. First, in such a case, it isn't necessary for the set of online parties $\mathcal{I}_{\text{on}}$ to be fixed ahead of time. Rather, the dealer P_i can prepare its message $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_i, s_j)$ for *all* $j \in [n]$ (including publicly-verifiable encryption for all parties) and the first $t-1$ parties to connect to the coordinator become the "online parties". Second, although communicating via such a coordinator machine makes no difference regarding the messages sent (since messages are signed they cannot be modified, and since they are encrypted the coordinator machine sees nothing), it does mean that the coordinator can block or erase messages at will. This means that the property that we desire – that if any honest online party accepts then so do all honest offline parties – cannot actually be achieved (specifically, the coordinator can refuse to deliver any valid message to an offline party). This means that the property achieved in this case is different and states that *if any honest online party accepts* ***and the coordinator is semi-honest*** *then all honest offline parties are guaranteed to accept.* However, once this is the case, the third round – where all parties exchange all signatures – can be avoided. Rather, the coordinator receives all $(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \sigma_j)$ at the end of round 2, and prepares the message $\left(\mathsf{share}, \mathsf{sid}, \mathcal{B}, \mathcal{V}, \{\sigma_k\}_{k \in \mathcal{I}_{\text{on}}}\right)$ which it sends to all offline parties. This therefore simplifies the protocol, and all online parties need to only connect once and then later download their output. We stress that if the ***coordinator is malicious***, then the only thing that can go wrong is for the parties to abort.

3.3 Adding a Party

We describe this protocol directly for t online parties, with the new party being added being a passive recipient only. The protocol requires each of the t parties to send and receive a message to the quorum, and to then prepare a final message for the new member. Thus, in the case of asynchronous parties, each party has to connect twice.

Idea: The idea behind this protocol is as follows. Let $\mathcal{I} \subseteq [n]$ be a set of online parties with $|\mathcal{I}| = t$, with respective shares $\{s_i\}_{i\in\mathcal{I}}$ on a polynomial $s(x)$. The aim of the parties is to generate $s(\alpha_{n+1})$ for the new party. By Lagrange interpolation, we have that $s(x) = \sum_{i\in\mathcal{I}} s_i \cdot L_i^{\mathcal{I}}(x)$ and thus $s(\alpha_{n+1}) = \sum_{i\in\mathcal{I}} s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1})$. Thus, each party P_i with $i \in \mathcal{I}$ can simply send the new party P_{n+1} the value $s_i^{n+1} = s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1})$, and P_{n+1} can compute the sum $s_{n+1} = \sum_{i\in\mathcal{I}} s_i^{n+1} = s(\alpha_{n+1})$. Unfortunately, this simple solution is insecure, since P_{n+1} can also compute

$$\sum_{i\in\mathcal{I}} s_i^{n+1} \cdot \frac{L_i^{\mathcal{I}}(0)}{L_i^{\mathcal{I}}(\alpha_{n+1})} = \sum_{i\in\mathcal{I}} s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1}) \cdot \frac{L_i^{\mathcal{I}}(0)}{L_i^{\mathcal{I}}(\alpha_{n+1})} = s_i \cdot L_i^{\mathcal{I}}(0) = s(0) = s$$

thereby revealing the secret itself. As a result, instead of each party directly sending $s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1})$ to P_{n+1}, the parties first subshare their shares amongst each other. Each party then locally sums the shares they receive, and the result is sent to P_{n+1}. With this method, P_{n+1} receives a random additive sharing of $s(\alpha_{n+1})$ which reveals nothing beyond that value, as required.

The simple way to implement this is for each P_i to generate *additive shares* of $s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1})$ to P_{n+1} to all of the participating parties. Each party can then sum up the sub-shares that it receives, and send to P_{n+1}. By the fact that $s_{n+1} = s(\alpha_{n+1}) = \sum_{i\in\mathcal{I}} s_i \cdot L_i^{\mathcal{I}}(\alpha_{n+1})$, it is immediate that P_{n+1} receives a random additive sharing of s_{n+1} as required. Our actual protocol works differently, since this would require that all parties know who the participating parties are *ahead of time*. In practice, we wish to enable the first t parties to connect to participate (asynchronously), and so we therefore have each P_i generate Shamir shares of s_i. By multiplying by the appropriate Lagrange coefficients, the same effect is achieved.

Protocol 3.6 (Π_{add})

Parties: *A set of t parties $\{P_i\}_{i\in\mathcal{I}}$ with $\mathcal{I} \subseteq [n]$, and a new party P_{n+1}*

Common input: $\mathsf{sid} \in \{0,1\}^*$, *parameters $t, n \in \mathbb{N}$ with $t \leq n$, and unique non-zero values $\alpha_1, \ldots, \alpha_{n+1} \in \mathbb{F}_q$*

P_i's private input: $(\mathcal{B}, s_i)$, *as output from $\mathcal{F}_{\mathsf{vss}}$*

(The protocol assumes that all parties have the same $\mathcal{B}$ and that s_i is the correct share as defined by $\mathcal{B}$. If this may not be the case, then parties need to begin by echo-broadcasting $\mathcal{B}$ (to ensure that all honest parties hold the same vector), and each party needs to locally verify that $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ where $\mathcal{B} = (B_0, \ldots, B_{t-1})$.)

The protocol:

1. ***Each party P_i subshares its share:***
   (a) *P_i chooses a random polynomial $s_i(x)$ of degree-$(t-1)$ such that $s_i(0) = s_i$*
   (b) *For every $j \in \mathcal{I}$, P_i computes $s_{i\to j} = s_i(\alpha_j)$*
   (c) *P_i sends $(\mathsf{sid}, s_{i\to j})$ to party P_j, for every $j \in \mathcal{I}$.*

   Note that if the set of participating parties $\mathcal{I} \subseteq [n]$ is not known ahead of time, then each P_i subshares to all parties, and the first t to connect continue to the next round

2. *P_i* ***generates the new party's subshare:*** *upon receiving $(\mathsf{sid}, s_{j\to i})$ from $t-1$ parties P_j*

   (a) *P_i computes $s_i^{n+1} = \sum_{j\in\mathcal{I}} s_{j\to i} \cdot L_j^{\mathcal{I}}(\alpha_{n+1})$*
   (b) *P_i sends $(\mathcal{B}, s_i^{n+1})$ to P_{n+1}*
   (If P_{n+1} is not online, then this message is encrypted under P_{n+1}'s public key, and signed with P_i private signing key, using secure signcryption.)

3. *P_{n+1} **prepares its output:** upon receiving t values $(\mathcal{B}, s_i^{n+1})$ from parties $\mathcal{I}$,*
   (a) *P_{n+1} verifies that all $\mathcal{B}$ values are the same from all parties, and aborts if not*
   (b) *P_{n+1} computes $s_{n+1} = \sum_{i\in\mathcal{I}} s_i^{n+1} \cdot L_i^{\mathcal{I}}(0)$*
   (c) *P_{n+1} verifies that $s_{n+1}\cdot G = \sum_{k=0}^{t-1}(\alpha_{n+1})^k \cdot B_k$, where $\mathcal{B} = (B_0,\ldots,B_{t-1})$, and aborts if not*
   (d) *P_{n+1} outputs $(\mathcal{B}, s_{n+1})$*

Correctness: Before proving security, we show (for the sake of clarity) that the protocol output is correct. Observe that

$$s_{n+1} = \sum_{i\in\mathcal{I}} s_i^{n+1} \cdot L_i^{\mathcal{I}}(0) = \sum_{i\in\mathcal{I}}\sum_{j\in\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) = \sum_{j\in\mathcal{I}} L_j^{\mathcal{I}}(\alpha_{n+1})\cdot \sum_{i\in\mathcal{I}} s_{j\to i}\cdot L_i^{\mathcal{I}}(0).$$

In the protocol, each $s_{j\to i}$ is generated by $s_{j\to i} = s_j(\alpha_i)$ where $s_j(0) = s_j$. Thus, $\sum_{i\in\mathcal{I}} s_{j\to i}\cdot L_i^{\mathcal{I}}(0) = s_j$ where $s_j = s(\alpha_j)$ is a point on the original sharing polynomial $s(x)$ for which $s(0) = s$ (and s is the original shared secret). Thus,

$$s_{n+1} = \sum_{j\in\mathcal{I}} L_j^{\mathcal{I}}(\alpha_{n+1})\cdot \sum_{i\in\mathcal{I}} s_{j\to i}\cdot L_i^{\mathcal{I}}(0) = \sum_{j\in\mathcal{I}} L_j^{\mathcal{I}}(\alpha_{n+1})\cdot s_j = s(\alpha_{n+1})$$

as required.

Security: We prove the security of the protocol under the assumption that all parties begin with consistent and valid input. That is, each party P_i is given input $(\mathcal{B}, s_i)$ where $s_i\cdot G = \sum_{k=0}^{t-1}(\alpha_i)^k\cdot B_k$ and $\mathcal{B} = (B_0,\ldots,B_{t-1})$, and all parties are given the *same* vector $\mathcal{B}$. This makes sense in practice since this input is the output of a previous VSS execution. However, as described in the protocol, if this may not be the case, then it needs to be separately verified. If the above holds, then we say that the input is consistent and valid.

Theorem 3.7 *Assume that the parties' inputs are consistent and valid. Then, protocol Π_{add} realizes functionality $\mathcal{F}_{\mathsf{add}}$ with perfect security-with-abort, in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t \le n$.*

Proof: Let $\mathcal{I} \subseteq [n]$ denote the set of online parties participating in the protocol, let $\mathcal{I}_c \subseteq \mathcal{I}\cup\{n+1\}$ denote the set of corrupted parties, and let $\mathcal{I}_h \overset{\text{def}}{=} \mathcal{I}\cup\{n+1\}\setminus\mathcal{I}_c$ denote the set of honest parties.[2] If $\mathcal{I}_c$ is empty, then simulation is trivial; we therefore assume that there is at least one corrupted party. We separately consider the case that the recipient (new party) P_{n+1} is honest, and the case that P_{n+1} is corrupted.

[2]In this protocol, only participating parties and P_{n+1} receive any messages. Therefore, we can ignore any other parties that may be corrupted.

Case 1 – the new party P_{n+1} is corrupted: The simulation in this case works by generating random values for all s_j^i values from honest parties, under the constraint that all values sum to s_{n+1}. This can be computed since s_{n+1} is given to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{add}}$ in this case of a corrupted P_{n+1}, and all corrupted s_j values are also known (by the assumption on valid and consistent inputs). Thus, it is possible to compute the sum of all the values the corrupted parties should send, and the sum of all values sent by the honest parties is just the difference between s_{n+1} and that sum. All the other messages can be honestly generated once these s_j^i values are computed.

Let $\mathcal{A}$ be the real-world adversary. We construct an ideal world adversary/simulator $\mathcal{S}$ as follows.

1. $\mathcal{S}$ sends $(\mathsf{add}, \mathsf{sid}, \mathcal{B}, s_j, \alpha_{n+1})$ to $\mathcal{F}_{\mathsf{add}}$ for every $j \in \mathcal{I}_c$
   (These inputs are consistent and valid by the assumption in the theorem.)
2. $\mathcal{S}$ receives $(\mathsf{add}, \mathsf{sid}, \mathcal{B}, s_{n+1})$ from $\mathcal{F}_{\mathsf{add}}$ (since P_{n+1} is corrupted)
3. $\mathcal{S}$ computes $s_h = s_{n+1} - \sum_{j\in\mathcal{I}_c} L_j^{\mathcal{I}}(\alpha_{n+1}) \cdot s_j$
4. $\mathcal{S}$ chooses random $\{s_j\}_{j\in\mathcal{I}_h}$ under the constraint that $\sum_{j\in\mathcal{I}_h\setminus\{n+1\}} s_j \cdot L_j^{\mathcal{I}}(\alpha_{n+1}) = s_h$
   (This implies that $\sum_{j\in\mathcal{I}} L_j^{\mathcal{I}}(\alpha_{n+1}) \cdot s_j = s_{n+1}$ and so the simulation will result in the correct output.)
5. $\mathcal{S}$ runs the honest parties in the protocol, following the exact protocol instructions but while using input $(\mathcal{B}, s_j)$ for honest party P_j
6. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs

The distribution over the messages seen by $\mathcal{A}$ is identical to a real protocol, since the input values chosen for the honest parties in the simulation sum up to the correct sum s_h (computed from s_{n+1} and the corrupted parties' inputs). These input values differ from the real inputs of the honest parties. However, since the honest parties subshare each share, including subshares amongst each other (between the honest parties), the distributions are identical. Formally, this holds because for every set of $s_{j\to i}$ messages from the honest parties to the corrupted parties, these can be the result of the real sharing or of the simulated sharing with the same probability, by adjusting a single message between two honest parties. Since there are at least two honest participating parties (because P_{n+1} is corrupted and a participating party is corrupted, at most $t-2$ participating parties are corrupted), such a message exists and is not seen by the adversary.

Case 2 – the new party P_{n+1} is honest: The simulation in this case works by just providing random $s_{j\to i}$ values from honest parties to corrupted parties. However, in this case, the simulator needs to determine if the honest P_{n+1} would output s_{n+1} or would abort. It can verify this by checking if the sum of all values sent by corrupted values equals what it is supposed to. We stress that the corrupted party may send incorrect values, but this doesn't matter since the honest party will output s_{n+1} *if and only if* the overall sum for all corrupted parties is correct. Let $\mathcal{A}$ be the real-world adversary. We construct an ideal world adversary/simulator $\mathcal{S}$, as follows:

1. $\mathcal{S}$ sends $(\mathsf{add}, \mathsf{sid}, \mathcal{B}, s_j, \alpha_{n+1})$ to $\mathcal{F}_{\mathsf{add}}$ for every $j \in \mathcal{I}_c$
   (These inputs are consistent and valid by the assumption in the theorem.)
2. *$\mathcal{S}$ sends the round 1 messages that $\mathcal{A}$ should receive from the honest parties:*

   (a) $\mathcal{S}$ chooses $s_{j\to i}$ randomly for all $j \in \mathcal{I}_h$ and $i \in \mathcal{I}_c$
   (b) $\mathcal{S}$ simulates each honest party P_j sending $s_{j\to i}$ to each corrupted party P_i

3. *$\mathcal{S}$ verifies that the sum of the messages sent by $\mathcal{A}$ is correct:*
   (a) Compute the sum of messages from $\mathcal{A}$:
      i. $\mathcal{S}$ receives from $\mathcal{A}$ messages $(\mathsf{sid}, s_{i\to j})$ for all $i \in \mathcal{I}_c$ and $j \in \mathcal{I}_h$; all the messages sent by corrupted parties to honest parties in round 1 of the protocol
      ii. $\mathcal{S}$ receives from $\mathcal{A}$ the messages $(\mathcal{B}, s_i^{n+1})$ for all $i \in \mathcal{I}_c$ that the corrupted parties send to the honest P_{n+1}
      iii. $\mathcal{S}$ computes $s_c = \sum_{j\in\mathcal{I}_c}\sum_{i\in\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) + \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot s_i^{n+1}$
   (b) Compute the sum of messages when playing corrupted parties honestly:
      i. $\mathcal{S}$ runs the corrupted parties $\mathcal{I}_c$ honestly, given the correct inputs $\{s_j\}_{j\in\mathcal{I}_c}$ and the first round messages $\{s_{j\to i}\}_{j\in\mathcal{I}_h, i\in\mathcal{I}_c}$ that it generated in Step 2b of the simulation above (and using fresh randomness for the corrupted parties)
      ii. Let $\{\hat{s}_{j\to i}\}_{j\in\mathcal{I}_c; i\in\mathcal{I}_h}$ be the round 1 messages and let $\{\hat{s}_i^{n+1}\}_{i\in\mathcal{I}_c}$ be the round 2 messages sent by the corrupted parties in the execution by $\mathcal{S}$ where it runs all corrupted parties honestly
      iii. $\mathcal{S}$ computes $\hat{s}_c = \sum_{j\in\mathcal{I}_c}\sum_{i\in\mathcal{I}_h} \hat{s}_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) + \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot \hat{s}_i^{n+1}$

4. *Completion of simulation:*
   (a) If $\hat{s}_c = s_c$ then $\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{add}}$ to provide the output to P_{n+1}; else, it instructs $\mathcal{F}_{\mathsf{add}}$ to provide abort / not provide output to P_{n+1}
   (b) $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs

It is immediate that the distribution over the messages $\{s_{j\to i}\}_{j\in\mathcal{I}_h, i\in\mathcal{I}_c}$ received by the corrupted parties from the honest parties is identical in the real and ideal executions. This holds since these values are less than t secret shares on a random degree-t polynomial. It remains to show that the honest party P_{n+1} outputs $(\mathcal{B}, s_{n+1})$ in a real execution if and only if it outputs $(\mathcal{B}, s_{n+1})$ in an ideal execution. Since the adversary can decide to cause an abort or not depending on the values sent by the honest parties in the first round, we also need to ensure that the joint distribution over the messages from the honest parties and whether or not P_{n+1} aborts or not is also identical. However, under the assumption (that we prove below) that $\mathcal{S}$ detects accurately whether or not P_{n+1} aborts, this follows from the fact that the adversary (with randomness fixed at the beginning of the execution) is fully determined by the honest parties' messages. Thus, the view of the adversary fully determines whether or not P_{n+1} aborts. We now conclude by showing that $\mathcal{S}$ accurately predicts if P_{n+1} would abort or not. Intuitively, this holds since once the honest parties' round 1 messages are fixed, the sum of what the corrupted parties send must be a fixed value. This is due to the fact that otherwise P_{n+1} would not receive the correct s_{n+1} (in which case it certainly aborts).

Formally:

$$\begin{aligned}
s_{n+1} &= \sum_{i\in\mathcal{I}} s_i^{n+1}\cdot L_i^{\mathcal{I}}(0)\\
&= \sum_{i\in\mathcal{I}}\sum_{j\in\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0)\\
&= \sum_{(j,i)\in\mathcal{I}\times\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0)\\
&= \sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) + \sum_{(j,i)\in\mathcal{I}_c\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0)\\
&\qquad + \sum_{(j,i)\in\mathcal{I}\times\mathcal{I}_c} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0).
\end{aligned}$$

Now,

$$\begin{aligned}
\sum_{(j,i)\in\mathcal{I}\times\mathcal{I}_c} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) &= \sum_{i\in\mathcal{I}_c}\sum_{j\in\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0)\\
&= \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot\sum_{j\in\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\\
&= \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot s_i^{n+1}
\end{aligned}$$

and so

$$s_{n+1} - \sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) = \sum_{(j,i)\in\mathcal{I}_c\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) + \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot s_i^{n+1}. \quad (1)$$

Observe that s_{n+1} is a fixed value (determined fully by the input sharing polynomial), albeit unknown to $\mathcal{S}$. In addition, $\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})$ is fully determined by the messages $s_{j\to i}$ sent by $\mathcal{S}$ to $\mathcal{A}$ in the simulation of the round 1 messages, albeit again unknown to $\mathcal{S}$. This holds because[3]

$$\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_c} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})$$

is fully determined by the messages sent from the honest parties to the corrupted parties in round 1 (by definition $\{s_{j\to i}\}_{(j,i)\in(\mathcal{I}_h\times\mathcal{I}_c)}$ is the set of all $s_{j\to i}$ messages sent by the honest to the corrupted). Next, we have

$$\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1}) = \sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1}) + \sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_c} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1}).$$

Thus, if $\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})$ is fully determined by the input values, then $\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})$ is fully determined by the input values and first message from the honest parties to the corrupted parties.

[3]This may seem obvious and therefore not require a proof. However, note that this sum refers to the messages sent *between the honest parties alone*, whereas the messages sent by $\mathcal{S}$ to $\mathcal{A}$ are from the honest parties to the corrupt parties.

In order to see that $\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})$ is fully determined from the input, observe that the t values $\{s_{j\to i}\}$ with $j\in\mathcal{I}_h$ and $i\in\mathcal{I}$ define a polynomial $s_j(x)$ with $s_j(0)=s_j$. This implies that for any $j\in\mathcal{I}_h$,

$$\sum_{i\in\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1}) = s_j(\alpha_{n+1})$$

and thus

$$\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1}) = \sum_{j\in\mathcal{I}_h} s_j(\alpha_{n+1})$$

which depends only on the input.

We conclude that the left-hand side $s_{n+1}-\sum_{(j,i)\in\mathcal{I}_h\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0)$ of Eq. (1) is *fixed and independent* of the messages sent by the adversary. This implies that after fixing the messages $\{s_{j\to i}\}_{(j,i)\in(\mathcal{I}_h\times\mathcal{I}_c)}$ from the honest parties to the corrupted parties in round 1, the simulator $\mathcal{S}$ can choose all random values $\{s_{i\to j}\}_{(i,j)\in\mathcal{I}_c\times\mathcal{I}}$ from the corrupted parties (independently of what $\mathcal{A}$ sends) and recompute

$$\sum_{(j,i)\in\mathcal{I}_c\times\mathcal{I}_h} s_{j\to i}\cdot L_j^{\mathcal{I}}(\alpha_{n+1})\cdot L_i^{\mathcal{I}}(0) + \sum_{i\in\mathcal{I}_c} L_i^{\mathcal{I}}(0)\cdot s_i^{n+1}$$

using the input values $\{s_i\}_{i\in\mathcal{I}_c}$ and these chosen random values, playing the corrupted parties "honestly". If this sum equals the sum of values received from the corrupted parties, then the output will be correct for P_{n+1} and so $\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{add}}$ to provide output. Otherwise, it instructs $\mathcal{F}_{\mathsf{add}}$ to not provide output to P_{n+1}. This completes the proof. $\square$

Observe that the simulator $\mathcal{S}$ in the proof of Theorem 3.7 is straight line (it does not rewind $\mathcal{A}$). Since perfect security with a straight-line simulator implies UC security, as proven in [8], we have the following corollary:

Corollary 3.8 *Assume that the parties' inputs are consistent and valid. Then, protocol Π_{add} UC realizes with abort the functionality $\mathcal{F}_{\mathsf{add}}$ in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t\le n$.*

3.4 Refresh

3.4.1 Secure Refresh in the $\mathcal{F}_{\mathsf{vss}}$-Hybrid Model

Intuition: The refresh protocol works by the parties generating a random secret sharing of 0 and adding it to the initial secret sharing. This clearly generates a random polynomial with the same constant term. In order to generate a random secret sharing of 0, we simply use $\mathcal{F}_{\mathsf{vss}}$, verifying that each secret sharing is of 0 (this is easy to do, since $s\cdot G$ is revealed, and so all that is needed is to verify that this equals the identity point $\mathcal{O}$).

Protocol 3.9 (Π_{refresh})

Parties: *A set of t online parties P_i with $i\in\mathcal{I}$*

Common input: sid $\in\{0,1\}^*$, *parameters $t,n\in\mathbb{N}$ with $t\le n$, and unique non-zero values $\alpha_1,\ldots,\alpha_n\in\mathbb{F}_q$*

P_i's **private input:** *$(\mathcal{B}, s_i)$, as output from $\mathcal{F}_{\mathsf{vss}}$*

(The protocol assumes that all parties have the same $\mathcal{B}$ and that s_i is the correct share as defined by $\mathcal{B}$. If this may not be the case, then parties need to begin by echo-broadcasting $\mathcal{B}$ (to ensure that all honest parties hold the same vector), and each party needs to locally verify that $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ where $\mathcal{B} = (B_0, \ldots, B_{t-1})$.)

The protocol:

1. ***Phase 1 (two rounds) – each party P_i deals a sharing of zero:***
   (a) *Each online P_i sends $\mathcal{F}_{\mathsf{vss}}$ the message $(\mathsf{share}, \mathsf{sid}\|i, 0)$*
   (b) *Every (online and offline) P_i receives $\left\{\left(\mathsf{share}, \mathsf{sid}\|j, \tilde{\mathcal{B}}_j, \tilde{s}_{j\to i}\right)\right\}_{i\in\mathcal{I}}$, where $\tilde{\mathcal{B}}_j = \left(\tilde{B}_0^j, \tilde{B}_1^j, \ldots, \tilde{B}_{t-1}^j\right)$ and $\tilde{s}_{j\to i}$ is P_i's private share in the secret sharing from P_j*
2. ***Phase 2 – each P_i generates output:***
   (a) *If not all t outputs $\left\{\left(\mathsf{share}, \mathsf{sid}\|j, \tilde{\mathcal{B}}_j, \tilde{s}_{j\to i}\right)\right\}_{i\in\mathcal{I}}$ are received (i.e., if any aborts), then output abort and halt*
   (b) *Let $\tilde{\mathcal{B}}_j = \left(\tilde{B}_0^j, \tilde{B}_1^j, \ldots, \tilde{B}_{t-1}^j\right)$ and let $\mathcal{B} = (B_0, \ldots, B_{t-1})$*
   (c) *Abort if any $\tilde{B}_0^j \neq \mathcal{O}$ (i.e., if any of the secret sharings are not to zero)*
   (d) *For $k = 0, \ldots, t-1$, set $B'_k = B_k + \sum_{j\in\mathcal{I}} \tilde{B}_k^j$*
   (e) *Set $\mathcal{B}' = (B'_0, B'_1, \ldots, B'_{t-1})$*
   (f) *Set $s_i = s_i + \sum_{j\in\mathcal{I}} \tilde{s}_{j\to i}$*
   (g) *Output $(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}', s_i)$*

We remark that Protocol Π_{refresh} can be instantiated with Π_{vss} or $\Pi_{\mathsf{vss}}^{\mathsf{off}}$ for $\mathcal{F}_{\mathsf{vss}}$, thereby yielding two variants with all online or only t parties online. This works since the parties only run local operations (to compute output) after the VSS output is received.

Security: We prove the security of the protocol under the assumption that all parties begin with consistent and valid input. That is, each party P_i is given input $(\mathcal{B}, s_i)$ where $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ and $\mathcal{B} = (B_0, \ldots, B_{t-1})$, and all parties are given the *same* vector $\mathcal{B}$. This makes sense in practice since this input is the output of a previous VSS execution. However, as described in the protocol, if this may not be the case, then it needs to be separately verified. If the above holds, then we say that the input is consistent and valid.

Theorem 3.10 *Assume that the parties' inputs are consistent and valid. Then, protocol Π_{refresh} realizes functionality $\mathcal{F}_{\mathsf{refresh}}$ in the $\mathcal{F}_{\mathsf{vss}}$-hybrid model with perfect security-with-abort, in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t \leq n$.*

Proof: Let $\mathcal{I}' \subseteq [n]$ denote the set of t online parties, let $\mathcal{I}_c \subseteq [n]$ denote the set of corrupted parties, and let $\mathcal{I}_h = [n] \setminus \mathcal{I}_c$ denote the set of honest parties. If $\mathcal{I}_c$ is empty, then simulation is trivial; we therefore assume that there is at least one corrupted party. Since not all parties are online and participate in the protocol, we denote by $\mathcal{I}'_c$ and $\mathcal{I}'_h$ the respective sets of online corrupted and honest parties (i.e., $\mathcal{I}'_c = \mathcal{I}' \cap \mathcal{I}_c$ and $\mathcal{I}'_c = \mathcal{I}' \cap \mathcal{I}_c$).

The idea behind the simulation is as following. The ideal functionality $\mathcal{F}_{\mathsf{refresh}}$ chooses a new random sharing, and provides the ideal adversary with its shares. The adversary can then choose to add any valid zero-sharing of its choice in order to determine the final polynomial. This reflects the fact that the real adversary can see the VSS shares of the honest parties before choosing its own shares. Thus, the sharing received by the ideal adversary/simulator $\mathcal{S}$ from $\mathcal{F}_{\mathsf{refresh}}$ reflects the sum of the original sharing and what the honest parties share. Thus, the simulator $\mathcal{S}$ subtracts the original sharing from the sharing received to receive a sharing of zero, and then simulates the honest parties sharings such that they sum to this sharing of zero. Finally, after receiving the corrupted parties' sharings from $\mathcal{A}$, the simulator sums them up and sends them as the polynomial to be added to the sharing by the ideal functionality.

We construct a simulator $\mathcal{S}$ as follows:

1. *Prepare values for simulating honest parties' sharings:*
   (a) $\mathcal{S}$ invokes $\mathcal{A}$ upon input $\{(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}, s_i)\}_{i\in\mathcal{I}'_c}$
   (These inputs are consistent and valid by the assumption in the theorem.)
   (b) $\mathcal{S}$ sends $\{(\mathsf{refresh}, \mathsf{sid}, \mathcal{B}, s_i)\}_{i\in\mathcal{I}'_c}$ to the trusted party computing $\mathcal{F}_{\mathsf{refresh}}$, and receives back $\left(\mathsf{refresh}, \mathsf{sid}, \{\hat{s}_i\}_{i\in\mathcal{I}_c}, \hat{\mathcal{B}}\right)$; let $\hat{\mathcal{B}} = (\hat{B}_0, \hat{B}_1, \ldots, \hat{B}_{t-1})$
   (c) $\mathcal{S}$ computes $\mathcal{B}_h = \left(\tilde{B}^h_0, \tilde{B}^h_1, \ldots, \tilde{B}^h_{t-1}\right)$, the public sum of the "honest parties' sharings", by $\tilde{B}^h_k = \hat{B}_k - B_k$ for $k = 0, \ldots, t-1$, where $\hat{B}_k$ is from $\hat{\mathcal{B}}$ received from $\mathcal{F}_{\mathsf{refresh}}$ and B_k is from $\mathcal{B}$ in the input
   (d) $\mathcal{S}$ computes $\{s^h_i\}_{i\in\mathcal{I}_c}$ the sum of the honest parties' shares sent to the corrupted parties by $s^h_i = \hat{s}_i - s_i$ for every $i \in \mathcal{I}_c$, where $\hat{s}_i$ is the new share received from $\mathcal{F}_{\mathsf{refresh}}$ and s_i is the corrupted party's previous share
   (Note that for every $i \in \mathcal{I}_c$ it holds that $s^h_i \cdot G = \sum_{k=0}^{t-1}(\alpha_i)^k \cdot \tilde{B}^h_k$.)
   (e) $\mathcal{S}$ chooses random $\{\tilde{s}_{j\to i}\}_{j\in\mathcal{I}'_h; i\in\mathcal{I}_c}$ under the constraint that for every $i \in \mathcal{I}_c$, $\sum_{j\in\mathcal{I}'_h} \tilde{s}_{j\to i} = s^h_i$
   (f) $\mathcal{S}$ finds random polynomials $\left\{\tilde{\mathcal{B}}_j\right\}_{j\in\mathcal{I}'_h}$ under the constraint that they sum to $\mathcal{B}_h$ and that for every $j \in \mathcal{I}'_h$ and $i \in \mathcal{I}_c$ it holds that $\tilde{s}_{j\to i} \cdot G = \sum_{k=0}^{t-1}(\alpha_i)^k \cdot \tilde{B}^j_k$, as follows:
      i. Let $j' \in \mathcal{I}'_h$
      ii. For every $j \in \mathcal{I}'_h \setminus \{j'\}$, $\mathcal{S}$ chooses a random degree-$(t-1)$ polynomial $\tilde{b}_j(x)$ such that $\tilde{b}_j(0) = 0$ and for every $i \in \mathcal{I}_c$ it holds that $\tilde{b}_j(\alpha_i) = \tilde{s}_{j\to i}$. $\mathcal{S}$ sets $\tilde{\mathcal{B}}_j = (\tilde{B}^j_0, \ldots, \tilde{B}^j_{t-1})$ by $\tilde{B}^j_k = \tilde{b}^j_k \cdot G$ where $\tilde{b}_j(x) = \sum_{k=0}^{t-1} \tilde{b}^j_k \cdot x^k$.
      (Note that $\mathcal{S}$ can always find such a polynomial. If $t-1$ parties are corrupted, then this polynomial is fully determined from $\tilde{s}_{j\to i}$ values and the fact that $\tilde{b}_i(0) = 0$.)
      iii. $\mathcal{S}$ computes $\tilde{S}_{j'\to i} = \tilde{s}_{j'\to i} \cdot G$ for $j' \in \mathcal{I}'_h$ specified above, and for every $i \in \mathcal{I}_c$. $\mathcal{S}$ then interpolates "in the exponent" to find a random polynomial $\tilde{\mathcal{B}}_{j'} = \left(\tilde{B}^{j'}_0, \ldots, \tilde{B}^{j'}_{t-1}\right)$ such that $\tilde{B}^{j'}_0 = \mathcal{O}$ (the additive identity) and for every $i \in \mathcal{I}_c$ it holds that $\tilde{S}_{j'\to i} = \sum_{k=0}^{t-1}(\alpha_i)^k \cdot \tilde{B}^{j'}_k$.
      (As above, $\mathcal{S}$ can always find such a polynomial since at most $t-1$ parties are corrupted and so at most t points are fixed, and since Lagrange interpolation can be computed "in the exponent" (i.e., on the group elements).)

2. *Simulate the refresh protocol:*

   (a) $\mathcal{S}$ simulates the protocol playing $\mathcal{F}_{\mathsf{vss}}$ using the $\tilde{B}_j$ and $\tilde{s}_{j\to i}$ values for all $j \in \mathcal{I}_h$ computed above (including aborting if any invalid messages are sent, as specified in the protocol and in the $\mathcal{F}_{\mathsf{vss}}$ description)

   (b) Let $\{(\mathsf{share}, \mathsf{sid}\|i, s_i(x))\}_{i\in\mathcal{I}'_c}$ be the messages the adversary $\mathcal{A}$ sends as the online corrupted parties sharings sent to $\mathcal{F}_{\mathsf{vss}}$
   (If not all sharings are sent, then $\mathcal{S}$ just waits.)

3. *$\mathcal{S}$ interacts with $\mathcal{F}_{\mathsf{refresh}}$:*

   (a) $\mathcal{S}$ computes $\tilde{s}(x) = \sum_{i\in\mathcal{I}'_c} s_i(x)$ and sends $\tilde{s}(x)$ to $\mathcal{F}_{\mathsf{refresh}}$ (the $s_i(x)$ polynomials are from the sharings sent by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{vss}}$ above)

   (b) $\mathcal{S}$ instructs $\mathcal{F}_{\mathsf{refresh}}$ to provide output to an honest party P_j if and only if $\mathcal{A}$ instructs $\mathcal{F}_{\mathsf{vss}}$ to provide output to P_j in *all* sharings of 0 from corrupted parties

4. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and halts

If the distribution over the polynomials $\left\{\tilde{\mathcal{B}}_j\right\}_{j\in\mathcal{I}'_h}$ and values $\{\tilde{s}_{j\to i}\}_{j\in\mathcal{I}'_h;i\in\mathcal{I}_c}$ is the same as in a real execution, then it is clear that the output distribution is identical. This is due to the fact that the rest of the simulation merely follows the instructions of the protocol, and due to the fact that the resulting sharing is guaranteed to match the output from the ideal functionality by how these values are chosen. However, since all $\tilde{s}_{j\to i}$ values are chosen at random under the constraint that they sum to the correct value, and since all other values are derived from these, the distribution of values seen by the adversary in the real and ideal executions is the same. Furthermore, the sum of the values chosen by the simulator match exactly the output received from the ideal functionality, as required. □

Observe that the simulator $\mathcal{S}$ in the proof of Theorem 3.10 is straight line (it does not rewind $\mathcal{A}$). Since perfect security with a straight-line simulator implies UC security, as proven in [8], we have the following corollary:

Corollary 3.11 *Assume that the parties' inputs are consistent and valid. Then, protocol Π_{refresh} UC realizes with abort the functionality $\mathcal{F}_{\mathsf{refresh}}$ in the $\mathcal{F}_{\mathsf{vss}}$-hybrid model in the presence of a static malicious adversary corrupting up to $t-1$ parties, for any $t \le n$.*

A note on the case that $t-1$ parties are corrupted: Observe that in this case, the refresh protocol is meaningless. This is due to the fact that since the polynomials shared by the honest parties go via $(0,0)$, the adversary can learn the exact polynomials shared. However, this is also true in the ideal model. In particular, given the new sharing and the old sharing, the adversary can compute the difference. This is a degree-$(t-1)$ polynomial with a zero constant term. Since here too the adversary has $t-1$ shares and the polynomial goes through $(0,0)$, the adversary with t points can interpolate to compute the polynomial itself. Thus, this same property holds in both the real and ideal models. Thus, there is no rerandomization of the polynomial sharing the secret here. Nevertheless, refresh is only of interest if at least two honest parties are honest, as shown in [1]. This is therefore not an issue.

Refresh with offline parties and a coordinator machine: In the case of asynchronous parties communicating via a coordinator machine that receives all (encrypted and signed) messages and forwards them to their appropriate destination, the refresh protocol can be implemented using $\Pi^{\mathrm{off}}_{\mathsf{vss}}$. This protocol guarantees that if any honest online party does not abort *and if the coordinator machine is semi-honest*, then all offline parties are guaranteed to not abort. However, in this protocol, each online party only needs to connect twice: in the first connection it plays the dealer, and in the second connection it receives the shares, signs the bundles and sends them to the coordinator. This can be a challenge for asynchronous parties, and we present a proposal for dealing with it in Section 3.4.2.

3.4.2 Refresh with ℓ Fully Online Parties, $t-\ell$ Asynchronous Online Parties, and $n-t$ Offline Parties

Aim: Our aim is to construct a refresh protocol that requires asynchronous online parties to only connect once, since connecting more than once for an operation that should happen regularly (like refresh) is very problematic. Clearly, if we assume that all participating parties are asynchronous, then this cannot be achieved. In particular, there is no way to ensure consensus between the values shared by the parties in a single round. However, if we assume that there are some fully online parties (we call them fully online parties to differentiate them from the asynchronous online parties who connect and disconnect), and we assume that at least one of these fully online parties is *semi-honest* then the consensus can be verified by them. We remark that the consensus could also be verified by the coordinator, and recall that in $\Pi^{\mathrm{off}}_{\mathsf{vss}}$ we also had the assumption that the coordinator was semi-honest. However, that protocol had the guarantee that if the coordinator was malicious, then the only bad thing that can happen is that parties abort. In contrast, here, if the coordinator verifies the consensus and it is malicious, then parties may output shares that are not consistent (i.e., do not lie on the same degree-$(t-1)$ polynomial). Therefore, rather than relying on a single machine, we assume that there are some ℓ fully online parties, and as long as *one of them is semi-honest – and not all of them malicious*, security (and even output delivery) is guaranteed.[4] We do stress, however, that if all fully online parties are malicious, then security is not guaranteed. (Although it is hard to separate correctness and privacy in secure computation, the aspect that breaks is only that of correctness. This means that the parties may end up having inconsistent shares, and so will not be able to operate. However, this can only happen if all fully online parties are corrupted.)

Protocol: The above discussion yields the following protocol for refresh.

Protocol 3.12 (Π'_{refresh})

Parties: *A set of ℓ fully-online parties $\{P_i\}_{i\in\mathcal{I}_{\text{perm-on}}}$, a set of $t-\ell$ asynchronous online parties $\{P_i\}_{i\in\mathcal{I}_{\text{async-on}}}$, and $n-t$ offline parties $\{P_i\}_{i\in\mathcal{I}_{\text{off}}}$. The sets $\mathcal{I}_{\text{perm-on}},\mathcal{I}_{\text{async-on}},\mathcal{I}_{\text{off}}$ are disjoint, and their union is $[n]$.*

Common input: $\mathsf{sid}\in\{0,1\}^*$, *parameters $t,n\in\mathbb{N}$ with $t\le n$, and unique non-zero values $\alpha_1,\ldots,\alpha_n\in\mathbb{F}_q$*

[4]As with the $\Pi^{\mathrm{off}}_{\mathsf{vss}}$, output delivery is guaranteed in the sense that if at least one of the honest online parties does not abort, then all offline honest parties do not abort.

P_i**'s private input:** $(\mathcal{B}, s_i)$, *as output from* $\mathcal{F}_{\mathsf{vss}}$

(The protocol assumes that all parties have the same $\mathcal{B}$ and that s_i is the correct share as defined by $\mathcal{B}$. If this may not be the case, then parties need to begin by echo-broadcasting $\mathcal{B}$ (to ensure that all honest parties hold the same vector), and each party needs to locally verify that $s_i \cdot G = \sum_{k=0}^{t-1} (\alpha_i)^k \cdot B_k$ where $\mathcal{B} = (B_0, \ldots, B_{t-1})$.)

The protocol:

1. ***Phase 1 (single round) – each asynchronous online party P_i shares a zero-sharing:***
   (a) *Each online P_i computes the first round of Protocol $\Pi^{\text{off}}_{\mathsf{vss}}$ for a sharing of zero, and with the offline parties defined to be all in $[n]$ (i.e., all parties' shares are encrypted using publicly-verifiable encryption)*
   (b) *Each online P_i signs their sharing values, and sends them to the fully-online parties*
2. ***Phase 2 – fully-online parties:***
   (a) *Each fully-online party runs rounds 2 and 3 of Protocol $\Pi^{\text{off}}_{\mathsf{vss}}$: the parties check all values and ZK proofs, echo-broadcast among themselves to verify consistency of all sharings, and gather signatures (of the fully-online parties and the asynchronous online party who shared)*
   (b) *Each fully-online party sends the bundle to the coordinator, for all parties to download*
3. *Output:*
   (a) *Each party (asynchronously-online or offline) downloads the bundle from the coordinator*
   (b) *Each party verifies that the bundle has the signature of all the fully-online parties, that the bundle contains t sharings of zero (i.e., verifying that each $B_0 = \mathcal{O}$), and that each sharing is signed by a different asynchronous-online party.*
   (c) *For each of the t sharings, each party decrypts their publicly-verifiable encryption and verifies its consistency with the sharing polynomial vectors $\mathcal{B}$*
   (d) *Each party runs phase 2 (generate output) in Π_{refresh}, and outputs the result*

Security: The proof of security of Protocol Π'_{refresh} is essentially the same as that of Theorem 3.10, with the only difference being which subset of parties carries out the consistency checks.

Acknowledgements

We thank Arash Afshar, Iftach Haitner and Samuel Ranellucci for helpful discussions.

References

[1] A. Afshar and Y. Lindell. Static Proactive Security and Share Refresh. Manuscript, 2024.

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

[3] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[4] J. Camenisch and I. Damgård. Verifiable Encryption, Group Encryption, and Their Applications to Separable Group Signatures and Signature Sharing Schemes. In *ASIACRYPT 2000*, Springer (LNCS 1976), pages 331-–345, 2000.

[5] P. Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In the *28th FOCS*, pages 427–437, 1987.

[6] M. Fischlin. Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors. In *CRYPTO 2005*, Springer (LNCS 3621), pages 152–168, 2005.

[7] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

[8] E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In the *SIAM Journal on Computing* (SICOMP), 39(5):2090–2112, 2010. (Preliminary version in the *38th STOC*, pages 109–118, 2006.)

[9] C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO 1989*, Springer (LNCS 435), pages 239–252, 1989.

[10] B. Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic. In *CRYPTO 1999*, Springer (LNCS 1666), pages 148–164, 1999.

[11] M. Stadler. Publicly Verifiable Secret Sharing. In *EUROCRYPT 1996*, Springer (LNCS 1070), pages 190–199, 1996.