COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services

Robin Vassantlal* Eduardo Alchieri† Bernardo Ferreira* Alysson Bessani*
*LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal †Universidade de Brasilia, Brazil

***Abstract*—Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) is a classical paradigm for implementing trustworthy services that has received renewed interest with the emergence of blockchains and decentralized infrastructures. A fundamental limitation of BFT SMR is that it provides integrity and availability despite a fraction of the replicas being controlled by an active adversary, but does not offer any confidentiality protection. Previous works addressed this issue by integrating secret sharing with the consensus-based framework of BFT SMR, but without providing all features required by practical systems, which include replica recovery, group reconfiguration, and acceptable performance when dealing with a large number of secrets. We present COBRA, a new protocol stack for *Dynamic Proactive Secret Sharing* that allows implementing confidentiality in practical BFT SMR systems. COBRA exhibits the best asymptotic communication complexity and optimal storage overhead, being able to renew 100k shares in a group of ten replicas $5\times$ faster than the current state of the art.**

***Index Terms*—Secret Sharing, BFT, Confidentiality**

I. Introduction

Context and motivation: Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) is a classical approach for implementing consistent and fault-tolerant services by deterministically executing the same sequence of commands in a set of replicas [1]. This approach maintains the *integrity and availability* of a secure service even if t out of the n replicas fail in an arbitrary/Byzantine way [2], [3]. BFT SMR gained a renewed interest with the emergence of blockchains [4], which can be seen as SMR systems that maintain an authenticated log of transactions [5], [6].

However, when considering the implementation of intrusion-tolerant services, sometimes called decentralized trusted third parties (T3P) [7], BFT SMR systems do not provide any means for securing the *confidentiality* of the service state in face of intrusions, e.g., if a single replica of a BFT SMR-based Key-Value (KV) store is compromised, the adversary can access all the data stored in the system.

Problem statement: We address the problem of designing practical BFT SMR services that can preserve the confidentiality of the stored data. In more detail, we are interested in protocols that can (i) operate in non-synchronous environments; (ii) deal with dynamic sets of participants in order to support replica servers crashing, recovering, leaving, and joining the system; (iii) support operations on thousands of secure data entries, as required in most real-world applications; and (iv) preserve their security properties (integrity, availability, and confidentiality) in long-lived systems, thus resisting strong adaptive adversaries that can eventually control a large number of nodes. Our ultimate goal is to be able to efficiently support large-scale applications with privacy and fault-tolerance requirements, such as blockchains and replicated KV stores.

Prior advancements: A strawman solution to provide confidentiality in BFT SMR is to store the data in an encrypted form. However, this would require a key distribution service if the stored data is shared among multiple clients. Implementing such a solution without relying on a trusted third party, requires another decentralized T3P, making the problem recursive. Some works avoid such recursion by storing data in clear on a subset of the servers, which are expected to not leak data (e.g., [8], [9]), albeit requiring more servers and stronger trust assumptions.

A more evolved solution is to enrich the service with *threshold cryptography*, in particular, a *secret sharing scheme* [10], [11]. Secret sharing protects the confidentiality of the stored data/secret by splitting it into pieces, called shares, in such a way that the secret can only be reconstructed by combining a fraction of its shares. Hence, each replica, instead of storing data in clear, stores a share of the secret (e.g., shares of values stored on a KV store). Therefore, if an adversary compromises a replica, it will only access the replica's share(s), which reveals nothing about the stored secret(s).

Starting with DepSpace [12], some works followed this approach to provide a confidentiality layer for practical BFT SMR services using *Verifiable Secret Sharing* (VSS) schemes [13]. In these schemes, besides the basic functionality described above, each share can have its integrity verified to cope with an active adversary that may corrupt it. A similar approach has also been used in more recent blockchain-oriented works [7], [14]–[16].

Nonetheless, those works do not consider an adaptive mobile adversary that can move from one replica to another and eventually collect the required number of shares to reconstruct the secret in a long-lived system. *Proactive Verifiable Secret Sharing* (PVSS, or simply PSS) schemes [17], [18] can be used to protect against such adversary by periodically refreshing the shares stored by the replicas without revealing the secret. Thus, if replicas are cleaned and rebooted [19]–[21], and shares are renewed before an adversary collects enough of them, the secrecy of the data will be maintained. Even if an adversary has collected some shares before a renewal, those shares are not useful as they cannot be combined with the renewed shares. Additionally, these schemes have also been extended to support a dynamic set of shareholders (DPSS - *Dynamic Proactive Secret Sharing*), either in a general setting [22], [23]

or considering blockchains [15], [16].

Despite a large body of work on secret sharing, there is still a fundamental limitation that prevents these schemes from being used in practical BFT SMR systems. Most secret sharing protocols deal with a single secret, however, in practice, these services (e.g., KV store) are expected to store thousands of records, each one comprising a different secret. This means that, when adding a new replica to the group or recovering a replica that lost its state, the shares of all these secrets need to be regenerated or recovered. Existing approaches have high communication and storage costs or are designed for the synchronous model, and thus are very inefficient or unsafe in non-synchronous environments. For instance, our implementation of MPSS [23] takes more than an hour for resharing 100 000 secrets in a group of ten servers (see §VIII).

Our solution: We address these limitations by presenting COBRA (COnfidential Byzantine ReplicAtion), a novel approach for proactive verifiable secret sharing in dynamic groups of processes. The centerpiece of COBRA is a *new protocol for distributed polynomial generation*. This protocol employs a Byzantine consensus algorithm to generate secret polynomials with shares distributed to at least $t + 1$ correct servers, which is enough for their reconstruction. These polynomials are used to transform the original shares during recovery and reshare [18].

The problem with this approach is that some executions can be adversary-influenced, and up to t correct servers may end up with invalid shares of the generated polynomial. This is not always an issue for share *recovery*, since $t + 1$ correct servers are enough to help a replica rebuild its state. However, secret *reshare* requires that all correct servers obtain valid renewed shares. COBRA solves this limitation in a natural but fundamentally different way than previous works by making servers that received invalid shares (during a resharing) execute the recovery protocol. The key challenge is then *how to make a process recover its share from a secret polynomial shared by only* $t + 1$ *correct processes*. Our solution is a novel recovery protocol that can identify and remove the processes that caused the generation of invalid shares during polynomial generation. In this way, correct servers can re-execute the recovery protocol, ignoring messages from removed faulty servers. In the worst case, this will lead to $t+1$ repetitions of the distributed polynomial generation, eventually reaching an execution in which only correct servers participate.

This interplay between the three COBRA protocols results in the best communication and storage complexity among existing non-synchronous DPSS protocols (see Table I), making it a solid foundation for supporting confidential BFT SMR with a large number of stored secrets. Using these protocols, we devised the first DPSS-based confidentiality framework for practical BFT SMR services, i.e., supporting replica recovery, secret resharing, and group reconfigurations.

We implemented COBRA on top of BFT-SMaRt [24], a popular BFT SMR library supporting all features needed in practical BFT SMR systems such as tolerance to asynchrony, crash recovery, and group reconfiguration. This implementation was evaluated through macro and micro benchmarks to assess the performance of COBRA with different numbers of secrets and replicas. The results show that COBRA is capable of processing thousands of 1kB-updates/second, being more than $3\times$ faster than the closest related work [7], and of performing recovery and resharing of 100 000 secrets using only 67% and 19% of the time required by the state-of-the-art protocols VSSR [7] and MPSS [23], respectively. For instance, COBRA requires less than 12 minutes for resharing 100 000 secrets in a group of ten servers, being $5\times$ faster than MPSS.

Contributions: In summary, we make the following contributions in this paper:

- We describe, for the first time, a general model for confidential BFT SMR that can accommodate various types of representative services (§III);
- We introduce the COBRA DPSS scheme – a modular protocol stack for distributed polynomial generation, share recovery, and dynamic secret resharing (§IV and §V) – achieving the best communication and storage complexity among existing schemes (§II);
- We implement COBRA DPSS and integrate it with the BFT-SMaRt replication library, making it the first system to provide all features required for practical BFT services while ensuring confidentiality (§VI and §VII). This implementation is available online [25];
- We present an experimental evaluation of COBRA considering up to 97 replicas and a state containing 100 000 stored secrets, showing it is significantly faster than existing related work. To the best of our knowledge, we are the first to evaluate the use of proactive secret sharing with such large number of secrets (§VIII).

II. Background and Related Work

A. Secret Sharing and its Variants

A *secret sharing* scheme transforms a *secret* s into n *shares* $s_1...s_n$ such that any combination of $t+1 \leq n$ of these shares can recover s, and no information about s can be obtained with t or less shares [10]. This mechanism considers three roles for processes: *dealer*, who distributes the secret; *shareholder*, who stores a share; and *combiner*, who recovers the secret using at least $t + 1$ shares. In the BFT SMR model considered in this paper (§III), the service clients play the 1st and 3rd roles, while the server replicas are shareholders.

The most popular implementation of secret sharing is the Shamir scheme [10]. In this scheme, the dealer builds a random polynomial P of degree t such that $P(0) = s$ and generates each of the n shares $s_1...s_n$ as points of P, i.e., $s_i = P(i)$. The secret s can be recovered by gathering $t + 1$ shares, interpolating the polynomial (i.e., recovering P), and calculating $P(0)$.

A key limitation of Shamir scheme is that it is not safe against an active adversary. More specifically, if some of the $t + 1$ shares used to interpolate the polynomial are corrupted, the generated polynomial $P' \neq P$ will lead to a different secret $s' = P'(0)$. To cope with this issue, the

TABLE I

COMPARISON OF PROACTIVE SECRET SHARING (PSS) SCHEMES. BEST-CASE COMMUNICATION AND STORAGE COMPLEXITY OF PROTOCOLS DEFINED IN TERMS OF n (NUMBER OF SHAREHOLDERS). WHEN REQUIRED, WE ASSUME THE SAME CONSTANT-SIZE COMMITMENT SCHEME [26] AND CONSENSUS PROTOCOL WITH LINEAR COMMUNICATION COMPLEXITY [27], [28] ARE USED IN ALL SCHEMES. ‡ CANNOT DECREASE THE POLYNOMIAL DEGREE.

PSS Scheme	Synchronous	Dynamic	Share/Combine	Reshare	Storage	Resilience
Herzberg et al. [18]	yes	no	$O(n)$	$O(n^3)$	$O(1)$	$t < n/2$
Cachin et al. [29]	no	no	$O(n^3)$	$O(n^4)$	$O(n)$	$t < n/3$
Desmedt and Jajodia [30]	yes	yes‡	$O(n)$	$O(n^2)$	$O(1)$	$t < n/2$
Wong et al. [31]	yes	yes‡	$O(n^2)$	$O(exp(n))$	$O(1)$	$t < n/2$
Baron et al. [32]	yes	yes‡	$O(n)$	$O(n^3)$	$O(1)$	$t < n/2 - \epsilon$
Opt-CHURP [15]	yes	yes‡	$O(n^2)$	$O(n^2)$	$O(n)$	$t < n/2$
Goyal et al. [16]	yes	yes	$O(n)$	$O(n^2)$	$O(1)$	$t < n/2$
Zhou et al. [22]	no	yes‡	$O(exp(n))$	$O(exp(n))$	$O(exp(n))$	$t < n/3$
MPSS [23]	no	yes‡	$O(n)$	$O(n^4)$	$O(n^2)$	$t < n/3$
Exp-CHURP [15]	no	yes‡	$O(n^2)$	$O(n^3)$	$O(n)$	$t < n/3$
COBRA DPSS (this paper)	**no**	**yes**	$\boldsymbol{O(n)}$	$\boldsymbol{O(n^3)}$	$\boldsymbol{O(1)}$	$\boldsymbol{t < n/3}$

scheme must be extended with commitments, implementing a *Verifiable Secret Sharing* (VSS) [13], [26], [33]–[36]. A (non-interactive) commitment scheme adds a piece of information to each share s_i, the commitment c_i, allowing shareholders and combiners to detect corrupted shares. For example, in the Feldman scheme [13], the commitment for a polynomial $P(x) = a_k x^k + ... + a_1 x + a_0$ is $c = \langle g^{a_k}, ..., g^{a_0} \rangle$, being g a public generator of a cyclic group, which reveals no information about the polynomial under the hardness of discrete log in multiplicative groups assumption. The verification that a point (x, y) is in a polynomial P (i.e., $P(x) = y$) is performed by checking if $g^y = (g^{a_k})^{x^k} \times ... \times (g^{a_1})^x \times (g^{a_0})$ is true.

A *Proactive Secret Sharing* (PSS) scheme can protect the secrecy of the data in a long-lived system against a mobile adversary that can eventually compromise more than t shareholders by periodically renewing the shares [17], [18]. For example, Herzberg et al. [18] introduced a PSS scheme[1] in which shareholders build a random *renewal polynomial* Q of degree t such that $Q(0) = 0$. The shares of the secret are renewed on each shareholder by adding the shares of Q to the shares of P, i.e., i uses $P(i)+Q(i)$ as its new share. The secret $P(0)$ will be preserved since $(P + Q)(0) = P(0)$. However, $P(i)$ (the old share) will be different from $(P + Q)(i)$ (the new share) since $Q(i) \neq 0$. This same scheme can also be used to recover the share s_i of a shareholder i. The idea is to use a *recovery polynomial* R such that $R(i) = 0$ to blind P, allowing i to recover the polynomial $P + R$ and evaluate $(P + R)(i) = P(i) = s_i$.

Proactive Verifiable Secret Sharing (PVSS) schemes [22], [29] combine the ideas above to support share-renewing and verifiability at the same time. Typically, this is done by generating commitments for all auxiliary polynomials and combining these commitments using additive homomorphism.

A key challenge in PVSS schemes is how to generate the auxiliary polynomial Q (or R, in case of recovery) in such a way that no party knows it entirely, only its share. In synchronous systems, this is easily done by making each party i generate a random polynomial Q_i and send its shares to every other party j. Every party waits for shares from all other parties for a fixed time and generates its share for the resulting polynomial by summing the received valid shares from different polynomials, i.e., $Q(j) = Q_0(j) + ... + Q_n(j)$ [18]. In asynchronous systems, this procedure becomes substantially more complicated since every party can only wait for at most $n - t$ other parties. In this case, the selection of the points to be used typically requires the execution of a *Byzantine consensus* protocol (e.g., [3], [28], [38]–[41]), either explicitly or by using a blockchain, to ensure that every party selects the same subset of polynomials to build Q [15], [23].

A natural extension of PVSS schemes is the support for changing the set of shareholders without endangering the secret's confidentiality. The key challenge here is to execute the resharing protocol with up to $2t$ malicious parties, t in the old group plus t in the new group. Works like MPSS [23], CHURP [15], Baron et al. [32], and Goyal et al. [16] deal with this problem using different techniques. These schemes are usually called *Dynamic Proactive Verifiable Secret Sharing* (DPVSS), or just DPSS for simplicity.

Table I summarizes some of the characteristics of existing schemes and COBRA DPSS, including the fault-free communication complexity of share/combine and reshare protocols, the storage overhead on each shareholder, and the resilience of the scheme. The table shows that there are only three schemes (besides COBRA) that support dynamism in non-synchronous systems, which is the typical setting in practical BFT SMR. Among these, Zhou et al. [22] requires an exponential number of messages on the number of shareholders. The other schemes are arguably more practical, but with different drawbacks. MPSS [23] has linear share/combine communication complexity (which is optimal), but has a high communication complexity on reshare due to the need to generate $n+1$ shared polynomials. CHURP [15] has a quadratic share/combine complexity due to the use of bivariate polynomials, in which each share is also a polynomial, but a cubic reshare complexity due to the use of this technique. Further, these protocols have a high storage overhead due to the need to store some messages from previous reshares [23] or due to the use of bivariate polynomials [15]. COBRA matches the optimal share/combine

[1]It is worth remarking that this scheme was shown to be vulnerable, but subsequent works patched it maintaining the same idea [37].

complexity of MPSS and the reshare complexity of (asynchronous) CHURP, while improving the storage overhead of both to $O(1)$ (also optimal). Moreover, COBRA can decrease the shared polynomial degree, following reconfigurations that decrease t, something previously available only on a synchronous scheme [16].

B. Data Confidentiality in BFT Systems

The seminal work on intrusion tolerance by Fraga and Powell [42] was the first to consider information scattering and threshold schemes for protecting data confidentiality in a replicated synchronous system. Later works like Secure Store [43] and CODEX [44] ensure confidentiality, integrity, and availability of stored data in asynchronous systems by using Byzantine quorum protocols [45] together with secret sharing.

To the best of our knowledge, DepSpace [12] was the first work to use secret sharing for achieving confidentiality in a BFT SMR system based on PBFT [3]. Similarly, Belisarius [46] also implemented a confidentiality-preserving storage service based on PBFT, with some tweaks for improving the performance of data reads. More recently, CALYPSO [14] proposed a practical blockchain-based infrastructure to store and share secrets. The two sharing protocols supported in CALYPSO are somewhat similar to what was done in DepSpace and CODEX, with some additional features, e.g., for better management of identities. Some of these systems [12], [46] achieved a performance similar to BFT SMR without confidentiality, but neither of them support the recovery of state, share renewal, or replica group reconfiguration, and thus are not adequate for practical BFT SMR systems.

The recent work of Basu et al. [7] partially solves this problem by introducing a share recovery protocol (VSSR) for *static* BFT SMR. However, the protocol uses *four additional recovery shares per stored secret*, which implies a significant overhead on storage, communication and computation (see §VIII). Similarly, some of the already mentioned works on asynchronous DPSS [15], [23] could be adapted to support reconfiguration on confidential BFT SMR, albeit with bad performance (see Table I).

Finally, it is worth mentioning the relationship between *Secure Multiparty Computation* (MPC) [47], [48] and confidential BFT SMR. In MPC, clients/parties jointly compute, typically in a synchronous peer-to-peer way, an arbitrary function without giving up their inputs' privacy. In BFT SMR, clients (asynchronously) submit commands to update the state of a service deployed on a set of untrusted servers. Therefore, confidential BFT SMR can be seem as a restricted form of MPC with temporal decoupling and support for restricted forms of private stateful computations. We expect to see more integration between these techniques as confidentiality solutions for BFT SMR mature. Additionally, since additively-homomorphic secret sharing is a fundamental building block for MPC, we believe COBRA DPSS might be of independent interest to recent MPC-as-a-Service [49] systems that consider long computations and dynamic participation [50].

III. Models and Goals

A. System Model

We consider a fully-connected distributed system composed by a universe of processes Π that can be divided into two subsets: an infinite set of replicas/servers $\Sigma = \{r_1, r_2, ...\}$, and an infinite set of clients $\Gamma = \{c_1, c_2, ...\}$. Clients access the BFT SMR system maintained by a subset of the replicas (a *view* - see next) by sending their requests to these replicas.

We assume a trusted setup, in which each replica and client has a unique identifier that can be verified by every other process of Π through standard means, e.g., through a public key infrastructure. More specifically, each replica r_i has a public-private key pair $\langle pk_i, sk_i \rangle$ used for signatures and encryption. A message m signed with sk_i is denoted by m_{σ_i}. A message m encrypted with pk_i is denoted by $E_i(m)$. We suppress process ids for readability when the involved processes are obvious.

We further assume a partially synchronous model [51] in which the network and processes may behave asynchronously until some *unknown* global stabilization time GST after which the system becomes synchronous, with *unknown time bounds for computation and communication*. This assumption is required to ensure the liveness of the consensus protocol employed by COBRA. Finally, every pair of processes communicate through *private and authenticated fair links*, i.e., messages can be lost and delayed, but not forever.

B. Dynamic Replica Groups and Views

We consider a dynamic system where BFT SMR replicas are able to join and leave the system during its execution by processing reconfiguration requests that *install a sequence of views* in the system [24], each one with the set of replicas at that time. These requests are totally ordered and, after a reconfiguration, all correct replicas will adopt the same *current view* $\mathcal{V}_{cur}$, which represents the most up-to-date view of the system. Until another view is installed, the replicas from $\mathcal{V}_{cur}$ are the only ones that may participate in the execution of client requests. We denote by $\mathcal{V}.n$ the number of replicas in a view $\mathcal{V}$ and $\mathcal{V}.t$ the number of replicas in $\mathcal{V}$ allowed to fail simultaneously. When not considering reconfigurations, we use just n and t for $\mathcal{V}_{cur}.n$ and $\mathcal{V}_{cur}.t$, respectively.

We further assume a non-empty initial view $\mathcal{V}_{ini}$ known to all processes. At any moment, clients obtain $\mathcal{V}_{cur}$ through standard means, e.g., by consulting a directory service [24].

C. Confidential SMR Service Model

The state machine replication model defines that all correct replicas work as a deterministic state machine and have the same state (the replicated *service state*) after executing the same sequence of operations [1]. However, by integrating secret sharing in this model, it becomes impossible for all replicas to have the same state, in particular because different replicas will have different shares of the stored secrets. Therefore, we define a *Confidential SMR* service model in which the system globally stores the state S, but locally each replica maintains two "states", one common to all replicas (similarly

to the standard SMR model) and one specific to each replica, containing the replica shares.

More formally, given a global state $S = \{D_1, ..., D_m\}$ composed of m *data entries*, each correct server r_i maintains a state $S_i = \langle C, P_i \rangle$, composed of two parts:

1) $C = \{\langle \overline{D}_1, D_1^e, c_1 \rangle, ..., \langle \overline{D}_m, D_m^e, c_m \rangle\}$ is the *common state*, represented by a set of tuples with the information replicated to all servers: $\overline{D}_j$ is the non-sensitive data associated with the data entry (e.g., the key associated with a value in a KV store, the metadata of a transaction in a blockchain), D_j^e corresponds to D_j encrypted using a random symmetric key K_j, and c_j is the commitment for the shares of K_j;
2) $P_i = \{s_{i,1}, ..., s_{i,m}\}$ is the *private state* of server r_i, containing its shares $s_{i,j}$ for each key K_j.

Additionally, the model considers that clients (acting as *dealers and combiners* w.r.t. shared secrets) manipulate data entries respecting access control policies implemented by replicas (the shareholders), as in other works (e.g., [12], [14]). Without such access control, a single corrupted client could trivially read all entries stored in the service.

This model explicitly incorporates secret sharing techniques for ensuring the confidentiality of the service state, allowing the specification of state recovery and reconfiguration protocols that require secret sharing primitives. Moreover, it is general enough to accommodate different types of services. In particular, we foresee three major types of services that can benefit from this model: (1) database services for the storage of small secret records (e.g., [14], [15], [44]), in which $\overline{D}_j$ is the public id of secret j, D_j^e is empty, and c_j and $s_{*,j}$ are the commitment and share of one of the records, i.e., secret sharing is applied directly over the small records; (2) services for the storage of large data blobs or files (e.g., confidential document databases with searchable encryption features [52]), where $\overline{D}_j$ can be a portion of the encrypted index, D_j^e is the encrypted data entry, and $s_{*,j}$ a share of the encryption key; and (3) outsourced encrypted databases allowing computations directly over the encrypted values [53], where D_j^e is, for instance, a homomorphically encrypted record and $s_{*,j}$ the share of its encryption key.

D. Adversary Model

We consider a *probabilistic polynomial-time (PPT) adaptive adversary*, that can control the network and may at any time decide to corrupt an unbounded number of clients and a fraction of the replicas in the current view. Corrupted clients and replicas are allowed to deviate arbitrarily from the protocol, i.e. they are prone to *Byzantine failures*. Such processes are said to be *faulty* or *corrupted*. A process that is not faulty is said to be *correct* or *honest*. More specifically, for a current view $\mathcal{V}$, the adversary can control simultaneously at most $\mathcal{V}.t = \lfloor \frac{\mathcal{V}.n-1}{3} \rfloor$ replicas. Additionally, since we consider that replicas erase their state after each view, the adversary can corrupt an arbitrary number of replicas that have left the system. For corrupted replicas, the adversary can learn the private state that they store.

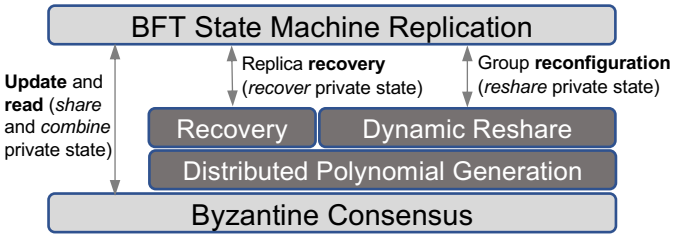

Fig. 1. COBRA protocol stack. Dark boxes represent COBRA DPSS.

For our proofs (Appendix B), we consider the recent definition of CC-adaptive security[2] [55], following the standard real/ideal paradigm.

E. Confidential SMR Security Goals

In addition to ensuring the standard *Safety* (or Linearizability [56], i.e., the replicated service emulates a centralized service) and *Liveness* (or Wait-freedom [57], i.e., all correct client requests are executed) properties of SMR [1], [19], COBRA also ensures a new property designated as *Secrecy*.

DEFINITION 3.1 (BFT SMR Secrecy): A dynamic BFT SMR system that evolves in a series of views $\mathcal{V}_{ini}, ..., \mathcal{V}_{cur}$ satisfies *Secrecy* of its state $S = \{D_1, ..., D_m\}$ if, for each view $\mathcal{V}$ and data entry $D_j \in S$ *not accessible by faulty clients*, an adversary controlling no more than $\mathcal{V}.t$ servers of $\mathcal{V}$ while $\mathcal{V}$ is the current system view, learns no information about D_j besides its non-sensitive part $\overline{D}_j$.

Secrecy ensures that no private information about the state of a confidential BFT SMR system can be obtained if the failure threshold of each current view is respected.

IV. COBRA Overview

COBRA aims to provide a confidentiality layer for practical BFT SMR systems. This is done through the protocol stack presented in Fig. 1. As in non-confidential BFT SMR systems, state updates and reads are ordered using a *Byzantine consensus* protocol. The difference here is that the shares $s_{1,j}, ..., s_{n,j}$ of the private part of data entry D_j will also be distributed (in updates) and obtained (in reads) together with the ordering of the request, with clients playing the roles of dealers and combiners, respectively.

Additionally, COBRA also uses Byzantine consensus for *distributed polynomial generation*. This protocol allows replicas to jointly create new random polynomials in a distributed way and serves as a basis for the two other protocols in the COBRA stack: replica recovery, and group reconfiguration.

Supporting replica recovery requires the re-generation of the replica's lost shares. When server r_i recovers from a failure, it needs (1) to obtain the common part C of its state (which is

[2]We resort to this model due to the well known "commitment problem" and our usage of efficient binding commitments (similarly to previous works [7], [15], [23]), which can not be properly simulated in the standard UC-adaptive model. Another possibility to deal with this problem would be to assume the usage of secure erasable memory [54], since we already consider that replicas erase their state between views.

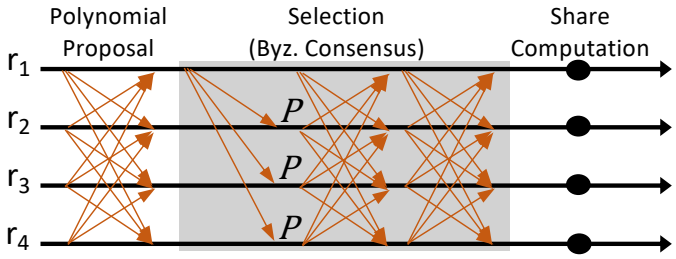

Fig. 2. Polynomial generation message pattern in an execution with $n = 4$.

replicated in other servers) using a state transfer protocol and (2) invoke the *share recovery* protocol for each stored data entry D_j, effectively reconstructing its private state.

Adding and removing replicas (group reconfigurations) can require changing the secret-encoding polynomials used in the service private state, e.g., reconfiguring a group $\mathcal{V}_i$ to a larger group $\mathcal{V}_{i+1}$ demands higher-degree polynomials to still tolerate $\lfloor (\mathcal{V}_{i+1}.n - 1)/3 \rfloor$ failures. This requires an operation for refreshing shares during group changes. COBRA supports a novel *dynamic reshare* protocol that needs to be invoked for each entry D_j on the state for transforming the shares $s_{1,j}, ..., s_{\mathcal{V}_i.n,j}$ into $s'_{1,j}, ..., s'_{\mathcal{V}_{i+1}.n,j}$.

Next we give a high-level overview of our protocols for share recovery and dynamic reshare, which are built on top of the distributed polynomial generation protocol. Together, these protocols form the COBRA DPSS scheme.

a) Distributed polynomial generation: This protocol allows a group of $n \geq 3t + 1$ servers to create a random polynomial P of degree t with an encoded point (x, y). At the end of the protocol execution, each correct server r_i obtains a share s_i of P and its commitment c.

This protocol, illustrated in Fig. 2, has three steps: (i) each server locally generates a random polynomial and distributes its shares to the group; (ii) servers run Byzantine consensus to ensure that correct servers select the same set with $t+1$ of these random polynomials; and (iii) the selected polynomials are summed, resulting in the shares of a polynomial P.

This protocol generates a shared random polynomial that is recoverable by correct servers, i.e., *at least* $t+1$ *correct servers* obtain a valid share of P, and *the other correct servers will detect that their shares are invalid* by using c. A key insight of COBRA is that this relatively simple protocol (when compared with previous works [15], [23]) is enough for supporting share recovery and reshare.

b) Share recovery: If a server r_k of $\mathcal{V}$ fails and recovers, it can reconstruct its shares through a share recovery operation. To recover its share s_k, r_k asks the other servers to generate a random recovery polynomial R, with $R(k) = 0$, using the distributed polynomial generation protocol. As discussed before, at least $t + 1$ correct servers will obtain valid shares of R, and use them to blind their own shares of P, generating shares of $P + R$, which are then sent to r_k. With $t + 1$ correct shares of $P+R$, r_k can recover its share by calculating $(P + R)(k) = P(k) + R(k) = P(k)$.

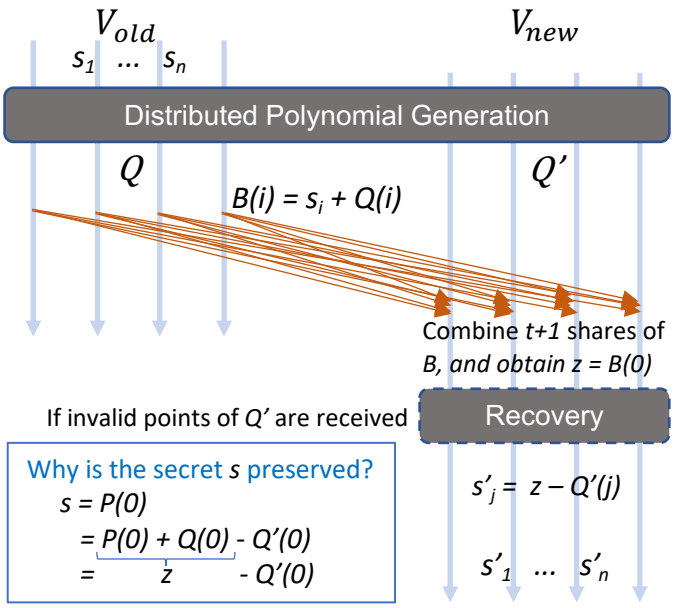

Fig. 3. COBRA dynamic reshare protocol.

This relatively simple procedure works only if all correct replicas have correct shares of P. More precisely, we have a set of correct servers S_P with shares of P and a set of correct servers S_R with shares of R. The described share recovery method works only if $|S_P \cap S_R| \geq t + 1$, i.e., when there are enough correct servers with shares of both P and R to answer r_k. However, in a BFT SMR scenario, a malicious client (dealer) can collude with faulty servers and distribute shares of a secret only to $|S_P| = t + 1$ correct servers and still execute its state update (see §VI). A similar situation can happen in the reshare protocol, as described next. The second novel insight of COBRA is how to execute a recovery even when not all correct servers have the required shares.

Our solution relies on the fact that correct servers that receive invalid shares of R at the end of the distributed polynomial generation know their shares are invalid and have an undeniable proof pointing to the faulty server b that misbehaved during the protocol execution. When this proof is revealed, correct servers start ignoring b and restart the protocol. In the end, the recovery can fail up to t times, removing t faulty servers, but eventually executing correctly.

c) Dynamic reshare: The dynamic reshare protocol allows a group of servers $\mathcal{V}_{old}$ to transfer the shared secret to a (possibly overlapping) group $\mathcal{V}_{new}$ by changing the shares while preserving the secret and its secrecy. Fig. 3 presents a high-level view of the protocol.

The first step of the protocol is to run a modified version of the distributed polynomial generation among the two (possibly overlapping) groups, generating two polynomials Q and Q', one for each group, encoding the same random secret on their independent term, i.e., $Q(0) = Q'(0)$. Servers in $\mathcal{V}_{old}$ blind their shares with Q and send them to $\mathcal{V}_{new}$. Each receiving correct server combines $t + 1$ blinded shares and calculates

the independent term of the blinded polynomial, $z = P(0) + Q(0)$. Using this value and their valid shares of Q', the correct servers of $\mathcal{V}_{new}$ obtain renewed shares that, when combined, lead to the same original secret.

Notice that a server of $\mathcal{V}_{new}$ can only obtain its share if it has a valid share of Q', something the distributed polynomial generation cannot guarantee for all correct servers. Therefore, when an invalid share is obtained, the server employs the share recovery protocol to get a valid share of Q', removing the malicious server(s) that was responsible for generating such invalid share from $\mathcal{V}_{new}$.

V. COBRA DPSS

We now detail COBRA's DPSS scheme, starting with its building blocks and functionality definition.

A. Building Blocks

1) Verifiable Secret Sharing (VSS): Our protocols employ a VSS scheme based on Shamir's work [10] to enable a client to share a secret among n servers. Formally, a VSS scheme provides the following functionality (omitting an Init protocol for simplicity):

- $\langle \{s_i\}_{i\in[n]}, c\rangle = \mathsf{Share}(n, t, x, y)$ is a randomized procedure that generates shares $\{s_i\}_{i\in[n]}$ and a commitment c of random polynomial P of degree t in which $P(x) = y$;[3]
- $false|true = \mathsf{Verify}(i, s_i, c)$ is a deterministic procedure that verifies if a share s_i is in polynomial P from which c was generated. If it returns $true$, we say s_i is *valid*;
- $y = \mathsf{Reconstruct}(S_{t+1}, x)$ is a deterministic procedure that obtains $y = P(x)$, where P is a polynomial of degree t interpolated using a set S of $t + 1$ valid shares.

The security of a VSS scheme lies in its *Hiding* and *Binding* properties [26]. *Hiding* states that an adversary cannot distinguish between two shared secrets with high probability, even if one is chosen by it and shared by the client through Share, and it has access to t shares of both. *Binding* states that the probability that an adversarial dealer can cause two distinct secrets to be reconstructed with the same commitment is negligible.

2) Byzantine Consensus (BC): We use a variant of BC to propose and decide values considered valid according to an application-defined predicate. The Consensus functionality can be described as follows. One or more correct processes in a view $\mathcal{V} \subset \Sigma$ *propose* a value v to all processes in $\mathcal{V}$. All correct processes in $\mathcal{V}$ will eventually *output a decision* v^* (*Termination*), which is the same (*Agreement*), and that was proposed by some $r_i \in \mathcal{V}$ (*Validity*). Additionally, we require *Validity* to be strengthened to ensure the decided value v^* is useful, i.e., it satisfy some application-specific predicate $\mathcal{P}$:

DEFINITION 5.1 ($\mathcal{P}$-Validity): A decided value $v*$ is valid (i.e., $\mathcal{P}(v^*) = true$) to at least $t + 1$ *correct* processes of $\mathcal{V}$.

[3]To simplify the exposition, we assume all shareholders use the same commitment c to validate their shares. In schemes where this is not true (e.g., [26]), we consider the part of the commitment specific to each share is sent together with it to its shareholder.

This property is a more general version of a similar property used in some consensus formulations [38], [58], [59]. These works consider a local predicate that can be computed in polynomial time, while our definition allows external factors to be used (e.g., the reception of some message that validates $v*$). The only requirement we make is that there is some value v among proposals such that eventually $\mathcal{P}(v) = true$ in every correct process of $\mathcal{V}$. For instance, this property can be easily implemented by extending a protocol like PBFT [3], as we do in COBRA. In this implementation, correct servers accept the leader proposal (PBFT's PRE-PREPARE message) only if the proposed value is valid w.r.t. to $\mathcal{P}$. Since a proposed value is decided in PBFT only if $2t+1$ (out of $3t+1$) servers accepted it, at least $t + 1$ correct servers need to accept it, satisfying *$\mathcal{P}$-Validity*. The same idea can be applied to other leader-based protocols such as HotStuff [28]. An interesting open question is if such property is implementable in more decentralized protocols (e.g., [38], [59]).

B. DPSS Definition

COBRA's Dynamic Proactive Secret Sharing definition extends the VSS with the Recover and Reshare functionalities. Shareholders use these to cope with failure recoveries and dynamism. They are specified as follows:

- $\langle s_k, c\rangle = \mathsf{Recover}(\mathcal{V}, k, \{s_i\}_{r_i\in\mathcal{V}\setminus\{r_k\}}, c)$ is a deterministic protocol that allows shareholders in $\mathcal{V}$ to collaboratively recover r_k's share s_k and corresponding commitment c using $\mathcal{V}.t + 1$ correct shares.
- $\langle \{s'_i\}_{r_i\in\mathcal{V}_{new}}, c'\rangle = \mathsf{Reshare}(\mathcal{V}_{old}, \mathcal{V}_{new}, \{s_i\}_{r_i\in\mathcal{V}_{old}}, c)$ is a randomized protocol that enables servers in $\mathcal{V}_{old}$ to move the secret to servers in $\mathcal{V}_{new}$, creating new shares $\{s'_i\}_{r_i\in\mathcal{V}_{new}}$ for them.

A DPSS scheme is considered secure with respect to the following definition [15], [23]:

DEFINITION 5.2 (DPSS): A secure DPSS scheme satisfies the following properties for any PPT adversary $\mathcal{A}$:

Secrecy: If $\mathcal{A}$ cannot corrupt more than $\mathcal{V}.t$ shareholders in the current view $\mathcal{V}$ during Recover, and more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$ during Reshare, then $\mathcal{A}$ learns no information regarding the shared secret.

Integrity: If $\mathcal{A}$ cannot corrupt more than $\mathcal{V}.t$ shareholders in the current view $\mathcal{V}$ during Recover, and more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$ during Reshare, then the shares for honest shareholders can be correctly computed and the shared secret remains intact.

Termination: In a partially synchronous model, if $\mathcal{A}$ cannot corrupt more than $\mathcal{V}.t$ shareholders in current view $\mathcal{V}$ during Recover, and more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$ during Reshare, then all honest shareholders eventually terminate these protocols.

Secrecy and *Integrity* ensure that the DPSS scheme is *secure*, while *Termination* states that the primitives are live.

Next, we detail the COBRA DPSS protocols.

C. Distributed Polynomial Generation

COBRA's distributed polynomial generation protocol allows a group of servers to create a random polynomial P of degree t encoding a point $(x, 0)$. This protocol serves as black-box component for the Recover and Reshare protocols, aiding in their execution.

More precisely, this functionality can be described as follows: $\langle S, \{s_i\}_{r_i \in \mathcal{V}}, c\rangle$ = GeneratePolynomial$(\mathcal{V}, x)$ is a randomized protocol that allows shareholders to generate a random polynomial P of degree $\mathcal{V}.t$ encoding $(x, 0)$. The function outputs at each correct server $r_i \in \mathcal{V}$ a share s_i of P, a commitment c, and a set S of polynomial proposals used to generate P. By the end of the protocol, at least $\mathcal{V}.t+1$ correct servers obtain valid shares of P, correct servers receiving corrupted shares can verify their shares are corrupted (using c), and the adversary $\mathcal{A}$ knows no more than $\mathcal{V}.t$ shares of P.

Our implementation of GeneratePolynomial is described bellow. The protocol has three phases, as illustrated in Fig. 2: *Polynomial Proposal* (Steps S1 and S2), *Selection* (Steps S3 and S4), and *Share Computation* (Step S5).

PROTOCOL GeneratePolynomial$(\mathcal{V}, x)$

Server $r_i \in \mathcal{V}$ with input $(\mathcal{V}, x)$:

- S1. Invokes $\langle\{s_{i,1}, ..., s_{i,n}\}, c_i\rangle$ = Share$(\mathcal{V}.n, \mathcal{V}.t, x, 0)$.
- S2. Sends $\langle$PROPOSAL$, E_1(s_{i,1})...E_{\mathcal{V}.n}(s_{i,\mathcal{V}.n}), c_i\rangle_{\sigma_i}$ to $\mathcal{V}$.
- S3. Collects $\mathcal{V}.n - \mathcal{V}.t$ PROPOSAL messages and runs Consensus:
  - a) Let $r_l \in \mathcal{V}$ be an elected consensus leader. r_l builds a set $S = \{\langle j, c_j\rangle : r_j \in \mathcal{V}\}$ using $\mathcal{V}.t + 1$ *valid proposals* (a proposal from r_j is *valid* for r_l iff $\langle$PROPOSAL$, ...E_l(s_{j,l})..., c_j\rangle_{\sigma_j}$ is correctly signed and both Verify$(x, 0, c_j)$ and Verify$(l, s_{j,l}, c_j)$ return *true*) and proposes it.
  - b) A correct replica accept the set S proposed by r_l if it received all proposals in S and they are valid (our $\mathcal{P}$-Validity predicate).
- S4. Waits for Consensus to output S. For all $\langle j, c_j\rangle \in S$, r_i asks for r_j's proposal $\langle$PROPOSAL$, ...E_i(s_{j,i})..., c_j\rangle_{\sigma_j}$ in $\mathcal{V}$ if it does not have it.
- S5. Decrypts all $E_i(s_{j,i})$, computes $s_i = \sum_{\langle j,*\rangle \in S} s_{j,i}$ and $c = \bigotimes_{\langle j,*\rangle \in S} c_j$, and outputs $\langle S, s_i, c\rangle$.

In the first phase servers create proposals with shares from a locally-created random polynomial and its commitment. All these shares are distributed to every server in an encrypted way, making all servers obtain all shares generated by correct servers. Further, proposals are signed by their creators to be used as an undeniable proof of misbehavior if needed.

In the second phase servers runs a Byzantine consensus to choose a set S of proposals that are valid for at least $t + 1$ correct servers. This works by having the consensus leader building S containing the identifiers of servers that sent valid shares to it and their associated commitments (which uniquely identifies the proposed polynomial). Due to *$\mathcal{P}$-Validity*, a correct server only accepts S if it received valid shares and matching commitments from all servers in S. Therefore, a decided set S corresponds to $t+1$ polynomial proposals with valid shares in at least $t + 1$ correct servers.

Notice that if a malicious server r_j sends a valid share to the leader r_l and invalid shares to other servers, and r_l proposes $\langle j, c_j\rangle$ in S, the consensus might not terminate with this proposal since S will not be accepted by enough servers. In this case, a timeout occurs and a new leader is selected [3]. When changing leaders, servers accuse r_j to the new leader using the invalid proposal sent by it. As a result, r_j's proposal will not be included in S, enabling consensus termination.

The final phase of the protocol is the computation of the shares and commitment of the generated random polynomial. This is done by adding the selected polynomials shares and combining their commitments (e.g., in Feldman's scheme, this is done by multiplying the commitments vector values [13]).

Communication complexity: In the proposal phase, each server sends n $O(n)$-size messages, resulting in a communication complexity of $O(n^3)$. The selection phase employs a Byzantine consensus protocol to select a set with $t + 1$ constant-size values. If a protocol with linear communication complexity like HotStuff [28] is used, the phase complexity would be $O(n^2)$. As a result, the fault-free communication complexity of the protocol is $O(n^3)$.

D. Share Recovery

To safely recover a server's share, we leverage the ideas of Herzberg et al. classical scheme [18], but adapted to non-synchronous environments. Given a polynomial P encoding a secret shared among a group of servers $\mathcal{V}$, the algorithm works differently depending on the existence of malicious servers and how many correct servers have shares of P. If all correct servers have such shares or if there are no malicious servers in $\mathcal{V}$, a single execution of the GeneratePolynomial is enough to generate a random polynomial R to blind the shares of P. Otherwise, there will be a set S_P of at least $t + 1$ correct servers with shares of P and a set S_R of at least $t+1$ correct servers with shares of R, with an intersection of only one correct server between these sets in the worst case. This is clearly not enough for recovery, since we need $t + 1$ correct servers with shares of both P and R.

Our solution is to identify and remove malicious servers that sent invalid proposals during the generation of R. Once they are removed (i.e., ignored forever), the protocol restarts. In the worst case, GeneratePolynomial can be repeated up to $t + 1$ times, to remove t malicious servers one by one, and then recover the share.

Bellow we present the protocol for recovering the share of a server (defined in §V-B). In the protocol, we mark the steps related with detection and removal of malicious servers in blue, to differentiate the expected normal case from the once-per-faulty-server repetition.

The recovery starts when r_k requests blinded shares to the other servers. When server r_i receives such request, it runs GeneratePolynomial to create a recovery polynomial R with degree t encoding $(k, 0)$. Each correct server that has a share of P and obtained a valid share of R calculates a blinded share with these two shares and return it to r_k. With $t + 1$ valid blinded shares, r_k recovers its original share.

PROTOCOL Recover$(\mathcal{V}, k, \{s_i\}_{r_i \in \mathcal{V} \setminus \{r_k\}}, c)$

Recovering server $r_k \in \mathcal{V}$ with input $\mathcal{V}$:

R1. Requests blinded shares from other servers in $\mathcal{V}$.
R2. Waits for responses.
- **a.** Upon receiving $\mathcal{V}.t + 1$ messages $\langle \mathsf{REC}, *, s^b_*, c, c^r \rangle$ (same c and c^r) with $\mathsf{Verify}(*, s^b_*, c \otimes c^r) = \mathit{true}$, outputs $\langle \mathsf{Reconstruct}(\{s^b_*\}, k), c \rangle$.
- **b.** Upon receiving a message $\langle \mathsf{ACC}, j, \mathit{prop}_j, s_{j,i} \rangle$, sends it to other servers in $\mathcal{V}$ and go to step R1.

Each server $r_i \in \mathcal{V} \setminus \{r_k\}$ with input $(\mathcal{V}, k, s_i, c)$:

S1. Upon receiving a blinded share request from r_k, runs $\mathsf{GeneratePolynomial}(\mathcal{V} \setminus \{r_k\}, k)$ to obtain $\langle S, s^r_i, c^r \rangle$.
S2. If the replica has a share for the secret being recovered, it evaluates $\mathsf{Verify}(i, s^r_i, c^r)$.
- a) If true, calculates $s^b_i = s_i + s^r_i$ and sends a message $\langle \mathsf{REC}, i, s^b_i, c, c^r \rangle$ to r_k.
- b) Otherwise, identifies the invalid proposal $\mathit{prop}_j = \langle \mathsf{PROPOSAL}, ...E_i(s_{j,i})..., c_j \rangle_{\sigma_j}$ of a server r_j in S and sends $\langle \mathsf{ACC}, j, \mathit{prop}_j, s_{j,i} \rangle$ to r_k.

S3. Upon receiving a message $\langle \mathsf{ACC}, j, \mathit{prop}_j, s_{j,l} \rangle$ from r_k, where $\mathit{prop}_j = \langle \mathsf{PROPOSAL}, ...E_l(s_{j,l})..., c_j \rangle_{\sigma_j}$ and $r_l \in \mathcal{V}$, checks if the accusation is sound (i.e., prop_j is signed by r_j, the encryption of $s_{j,l}$ with r_l's public key matches $E_l(s_{j,l})$, and $\mathsf{Verify}(l, s_{j,l}, c_j) = \mathit{false}$). If yes, adds r_j to an ignore list, otherwise includes r_l in such list.

If r_i receives an invalid share in GeneratePolynomial, it identifies the invalid proposal(s) that caused this and returns an accusation to r_k. When r_k receives this accusation, it disseminates it to all servers and restart the recovery. Upon receiving an accusation, server r_i verifies its soundness (i.e., the proposal is signed and contains an invalid share). If this is the case, the proposal producer is added to an ignore list, otherwise, the sender of the invalid accusation goes to this list.

Communication complexity: Apart from the share request and its reply, the only other communication of the protocol is due to the distributed polynomial generation, which has communication complexity of $O(n^3)$. Therefore, the fault-free communication complexity of the recovery protocol is $O(n^3)$.

E. Dynamic Reshare

COBRA's DPSS protocol for dynamic reshare leverages ideas from classical and recent works on the topic [16], [18]. However, differently from these works, we assume a non-synchronous setting, which significantly complicates the realization of the approach. For instance, if an adversary controls some servers in $\mathcal{V}_{old}$, we might require an execution of the recovery protocol to remove such servers and create a renewed share for every correct server of $\mathcal{V}_{new}$.

We presented a high-level view of the reshare protocol in Fig. 3. The key idea is to generate two random helper polynomials Q and Q', such that $Q(0) = Q'(0) = q$, being q a random factor used for blinding the secret. The shares of Q are used by servers in $\mathcal{V}_{old}$ to blind their stored shares. Servers in $\mathcal{V}_{new}$ use Q' shares and the blinded secret z (obtained using the blinded shares from $\mathcal{V}_{old}$) to compute their renewed shares.

In the following, we detail how our protocol reshares a secret, while "moving" shares from $\mathcal{V}_{old}$ to $\mathcal{V}_{new}$. The protocol is divided in three phases: *Polynomial Generation* (Steps O1, N1, and N2), (2) *Blinded Secret Reconstruction* (Steps O2 and N3) (3) *Share Renew* (Steps O3 and N4).

PROTOCOL Reshare$(\mathcal{V}_{old}, \mathcal{V}_{new}, \{s_i\}_{r_i \in \mathcal{V}_{old}}, c)$

Each server $r_i \in \mathcal{V}_{old}$ with input $(\mathcal{V}_{old}, \mathcal{V}_{new}, s_i, c)$:

O1. Executes GeneratePolynomial$*$ to obtain Q and Q' for $\mathcal{V}_{old}$ and $\mathcal{V}_{new}$, respectively, and obtains $\langle S, s^q_i, c^q \rangle$.
O2. If $\mathsf{Verify}(i, s^q_i, c^q)$ is true, calculates a blinded share $s^b_i = s_i + s^q_i$ and its commitment $c^b = c \otimes c^q$, and sends $\langle s^b_i, c^b \rangle$ to $\mathcal{V}_{new}$.
O3. Deletes $\langle s_i, c \rangle$ and halts.

Each server $r_j \in \mathcal{V}_{new}$ with input $(\mathcal{V}_{old}, \mathcal{V}_{new})$:

N1. Executes GeneratePolynomial$*$ to obtain $\langle S, s^*_j, c^{q'} \rangle$.
N2. If $\mathsf{Verify}(j, s^*_j, c^{q'}) = \mathit{false}$, runs Recover to obtain a valid share $s^{q'}_j$ of Q'.
N3. Waits the reception of $\mathcal{V}_{old}.t + 1$ messages $\langle s^b_*, c^b \rangle$ with the same c^b and $\mathsf{Verify}(*, s^b_*, c^b) = \mathit{true}$, and calculates $z = \mathsf{Reconstruct}(\{s^b_*\}, 0)$.
N4. Compute the renewed share $s'_j = z - s^{q'}_j$ and its corresponding commitment $c' = c^b \odot c^{q'}$. Outputs $\langle s'_i, c' \rangle$.

The protocol starts with all servers invoking the GeneratePolynomial$*$ protocol, a *modified version* of GeneratePolynomial that generates *at once* two random polynomials Q and Q' with degree $\mathcal{V}_{old}.t$ and $\mathcal{V}_{new}.t$, respectively, encoding the same random secret value q at their independent terms. Apart from these differences, GeneratePolynomial$*$ implements a functionality similar to what was described in §V-C, and has the same three phases and five steps of GeneratePolynomial. We defer the specification of this protocol to Appendix A.

After executing GeneratePolynomial$*$, it might happen that some correct servers of both groups receive invalid shares of Q or Q'. This is not a problem for $\mathcal{V}_{old}$ since at least $\mathcal{V}_{old}.t + 1$ correct server will have shares of Q, which is enough to generate a blinded secret. However, servers in $\mathcal{V}_{new}$ need valid shares of Q' to (re)build their shares of the secret, so they may need to run the Recover protocol.

In the second phase of the protocol, servers in $\mathcal{V}_{old}$ compute *blinded shares* and send them to $\mathcal{V}_{new}$ servers, which use them to reconstruct the blinded secret z. After that, in the last phase, correct servers in $\mathcal{V}_{new}$ use z and their shares of Q' to compute their renewed shares and the corresponding commitment.[4]

Communication complexity: Assuming a constant commitment scheme, the second phase of the protocol requires all $\mathcal{V}_{old}$ servers to send constant-size messages with their blinded shares and commitments to $\mathcal{V}_{new}$ servers, leading to a quadratic asymptotic communication. As a result, the fault-free communication complexity of resharing is dominated by GeneratePolynomial$*$, resulting in $O(n^3)$, as in Table I.

[4]In this step, $\odot$ represents the inverse operation of $\otimes$, e.g., in Feldman's scheme, this is a division of the commitments vectors values.

F. Security Analysis

COBRA DPSS is secure since it fulfills the *Secrecy*, *Integrity* and *Termination* properties of DPSS (Definition 5.2). We present a proof sketch of this claim in Appendix B.

VI. From DPSS to BFT SMR

The COBRA protocol stack for confidential BFT SMR enables fundamental features typically needed in practical deployments, such as replica recovery and group reconfiguration. In the following we outline how we use the COBRA DPSS scheme for supporting confidential BFT SMR. We discuss the correctness of our design in Appendix C.

A. State Update and Read Protocols

Recall that the confidential SMR service model considers that each server's state is composed by a common state and a private state (see §III-C). The common part is updated (resp. read) using standard BFT SMR, i.e., employing Byzantine consensus to ensure correct replicas processes client-issued requests in total order [3], [58], while the private part requires the distribution (resp. collection) of the shares. Clients act as dealers/combiners that distribute/collect the shares to/from the replicas by using the previously defined Verifiable Secret Sharing (VSS) functions (see §V-A). COBRA employs request processing protocols similar to the ones described in [7].

The *update* functionality $\langle\langle\overline{D}, D^e, c\rangle, \{s_i\}_{r_i\in\mathcal{V}}\rangle = \mathsf{Update}(\mathcal{V}, D)$ is implemented by a randomized protocol that uses Share and Verify to distribute shares s_* of confidential data D and Byzantine consensus to multicast its associated common data $\langle\overline{D}, D^e, c\rangle$ to servers in $\mathcal{V}$. The key challenge in implementing this protocol is to synchronize the reception of these two pieces of information in a single atomic update. This is guaranteed due to Byzantine consensus' $\mathcal{P}$-*Validity* property, which ensures a client-issued operation is accepted by a replica only if it receives the corresponding valid share from the client. This guarantees that an update implies having at least $t+1$ correct replicas with valid shares of the associated private state, which is enough for other correct replicas to recover their shares in case of need.

The *read* functionality $D = \mathsf{Read}(\mathcal{V}, \overline{d})$ is implemented by a deterministic protocol that retrieves the confidential data associated with $\overline{d}$ from servers in $\mathcal{V}$, and uses Verify and Reconstruct to compute the correct confidential data D. To read entries from the confidential state, clients need to obtain (together with the common part of the entries) at least $t+1$ valid shares for the entries.

B. State Recovery

In practical BFT SMR systems, it is expected that servers will fail and lose their states. The *state recover* functionality $\langle C, P_k\rangle = \mathsf{StateRecover}(\mathcal{V}, k, C, \{P_i\}_{r_i\in\mathcal{V}\setminus\{r_k\}})$ is implemented by a deterministic protocol that reconstructs the state of a replica $r_k \in \mathcal{V}$. Since the common state is the same for all servers, C is retrieved using a traditional SMR state transfer protocol [3], [60], [61]. The private state $P_k = \{s_{k,1}, ..., s_{k,m}\}$ is recovered through multiple executions of the DPSS Recover protocol (see §V-D).

The most complicated part of our implementation of this protocol consists in running multiple instances of Recover, which implies in the distributed generation of m polynomials for blinding different shares. This can be done in three ways: by running m instances of the protocol, by running a single instance of the protocol but adapted to generate m polynomials at once, or by mixing these two, i.e., running u instances of the protocol, each one generating w polynomials, such that $m = u \times w$. We use this last approach, with a configurable w.

Notice a server does not need to wait for all shares to be recovered before starting to process client-issued operations. Once the common state is recovered using standard SMR methods, operations that require only the state already recovered can be processed. Furthermore, it is possible to prioritize the recovery of the shares of certain data items based on their popularity to enable the recovering replica to serve such items as soon as possible. There is a rich literature on how to do this prioritization (mostly for database sharding) that can be adapted to COBRA (e.g., [62]).

C. Group Reconfiguration

Service reconfigurations are typically executed by agreeing on a new set of system replicas [24]. The *reconfigure* functionality $\langle C', \{P'_i\}_{r_i\in\mathcal{V}_{new}}\rangle = \mathsf{Reconfigure}(\mathcal{V}_{old}, \mathcal{V}_{new}, C, \{P_i\}_{r_i\in\mathcal{V}_{old}})$ is a randomized protocol that enables BFT SMR service to change its configuration from $\mathcal{V}_{old}$ to $\mathcal{V}_{new}$. C' is the new common state stored by each replica in $\mathcal{V}_{new}$, and only differs from C in its polynomial commitments. When a reconfiguration does not require removing replicas or modifying t (e.g., a group with four servers in which two more are added, keeping $t = 1$), joining replicas can obtain the state using the StateRecover protocol described in previous section. Otherwise, i.e., when changes in system composition imply replica removals or changing t, COBRA DPSS Reshare protocol (see §V-E) needs to be executed for each secret kept in the system. Just as in state recovery, refreshing the shares may require the execution of the resharing protocol multiple times, each time for several secrets, and can start serving some data items as soon as they are refreshed.

Reconfigurations can also be used to cope with mobile adversaries [17], [18]. The idea is to execute a "fake" reconfiguration, without adding or removing replicas, and hence generating a new view in which all servers reshare their secrets and make shares from the previous view invalid. This mechanism is important for supporting proactive recovery [19]–[21].

D. Optimizations

1) Speeding up share verification: The cost of share verification affects the system scalability since recovering or resharing a large number of shares can become the dominant performance factor. To reduce the impact of such verification, we combine the Harn et al. detection scheme [35] with a traditional commitment scheme. The former allows detecting the

existence of at most t invalid shares during the interpolation of a polynomial. In case there is a problem, we use the latter to identify the invalid shares and their senders, which are ignored when collecting shares for future interpolations.

The detection scheme requires at least $t+2$ shares, instead of $t+1$, to detect up to t invalid shares [35]. The rationale is that the interpolation of a polynomial using more than $t+1$ shares will have degree $d=t$ if all the used shares are valid. Otherwise, we will have $d>t$ with overwhelming probability.

Since the recovery of a secret or a share requires polynomial interpolation, it is possible to detect invalid shares just by waiting for one more share. Thus, the costly commitment scheme is used to identify invalid shares and their senders *only when the detection scheme identifies the existence of an invalid share*. Since malicious servers are ignored after identification, the commitment scheme ends up being used only during the interpolation of at most t polynomials. Therefore, multiple interpolations can be performed efficiently, decreasing the time taken to reconstruct a secret and, more importantly, scaling recover and reshare for large private states.

2) Decreasing share recovery cost: During the execution of the polynomial generation protocol, honest servers collectively select $t+1$ valid polynomials. However, the sum of these polynomials generates a single secret polynomial (Step S5 of GeneratePolynomial), wasting t good polynomials. This squandering can be avoided by integrating a Vandermonde matrix into the distributed polynomial generation, potentially increasing the throughput by a factor of t. This technique, which implements a form of batching [32], allows the transformation of $t+1$ selected polynomials into $t+1$ random polynomials [16], [63].

The application of this technique also requires transforming commitments, which has a higher computational cost. Therefore, it is not beneficial for the Reshare protocol. However, we can leverage the Vandermonde matrix to increase the recovery polynomial generation throughput since the commitments are only used by the recovering server if there is a corrupted blinded share.

VII. Implementation

We implemented COBRA in Java on top of BFT-SMaRt [24], a replication library providing all features required for practical BFT SMR systems.

Byzantine consensus: BFT-SMaRt implements a Verifiable and Provable Consensus [58] based on Cachin's Byzantine Paxos [39], which is similar to PBFT [3]: requires three communication steps and has a quadratic message complexity. This is the consensus algorithm used in our implementation of the distributed polynomial generator.

Cryptographic primitives: We implemented Shamir's secret sharing scheme using Lagrange polynomial interpolation and Horner's method [64] for share computation. This method ensures that a polynomial of degree t can be evaluated using only t multiplications and t additions, instead of t exponentiations, t multiplications, and t additions. All secret sharing computations were done in a prime field of 256 bits.

COBRA supports two commitment schemes. Feldman's *linear* commitment [13], which comprises a vector of t integers, and Kate et al. commitment [26], which uses a *constant* number of integers and is implemented using *optimal ate pairing* over Barreto-Naehrig curve. Additionally, pairing-related operations are performed using the RELIC library [65]. We use prime fields of 2048 and 256 bits for the Feldman and Kate et al. schemes, respectively.

We implement the system to support the execution of multiple instances of COBRA protocols in parallel. More specifically, the cryptographic processing for different instances can be executed in parallel by a number of configurable threads. Further, we use `SHA256` for hash functions, `SHA256withECDSA` for signatures, TLS without a cipher suite for implementing authenticated channels (`TLS_ECDHE_ECDSA_WITH_NULL_SHA`), and `AES` with 256-bit keys for encrypting confidential information carried on messages (e.g., shares).

Confidential Key-Value Store: We evaluate COBRA using a KV store that can be used to store confidential data. This is a simple application used to evaluate many BFT SMR systems [60], [66], [67], including some with confidentiality [7], [46]. The idea is to support a put/get interface in which pairs $\langle k, v\rangle$, such that k is an open key and v is a secret that should not be revealed if a replica is compromised, are written and retrieved from the system.

Other protocols: Besides COBRA, we also implemented two other recovery and resharing protocols to experimentally compare our approach with the state of the art:

1) VSSR [7]: A recent approach for share recovery in which a constant number of recovery polynomials are created together with the secret to be shared.
2) MPSS [23]: The previously-known most efficient protocol for dynamic resharing in non-synchronous BFT SMR systems.[5] MPSS employs a Byzantine consensus protocol (PBFT [3]) to make servers agree on the set of contributors for generating renewal polynomials, requiring thus a single consensus execution per reshare, just like COBRA.

Open-Source Implementation: Our COBRA implementation extending BFT-SMaRt and all the code used for the experiments in this paper (see next section) are available on the project web page [25].

VIII. Evaluation

We performed an experimental evaluation of COBRA and competing protocols, aiming at answering the following questions: *What is the overall impact of integrating COBRA's DPSS into BFT SMR normal operation? What is the cost of recovering and resharing a significant number of secrets using*

[5]We did not considered CHURP because, although asymptotically better than MPSS in Table I, the use of a quadratic consensus protocol (as in BFT-SMaRt) would lead CHURP to have the same bit complexity of MPSS ($O(n^4)$). Furthermore, the fact that CHURP requires $O(n^2)$ consensus executions per reshare [15] and its quadratic share/combine communication complexity, would make it inefficient for BFT SMR.

COBRA? What are the dominating factors on the latency of the proposed protocols? How much does the presence of corrupt replicas impacts the latency of COBRA protocols?

A. Setup and Methodology

All experiments were executed in a cluster composed of 14 physical machines connected through a gigabit ethernet. All machines are Dell PowerEdge R410 servers, with 32GB of memory and two quadcore 2.27 Intel Xeon E5520 processor with hyperthreading (supporting thus 16 hardware threads). The machines run Ubuntu Linux 20.04.1 LTS and JRE 1.8.

We present two types of experiments. First, a set of *macro benchmarks* based on a KV store deployed on top of COBRA. For these experiments, we consider small groups of up to ten replicas ($t \leq 3$), each deployed in a physical machine, with four other machines running up to 1500 clients issuing transactions to the system. Second, a set of *micro benchmarks* used to evaluate the cost of certain steps of the protocols, considering a large number of replicas and secrets. As in previous works (e.g., [7], [15], [16]), our experiments focus on *fault-free executions of the protocols* to assess the overhead imposed by a secret sharing layer in typical executions of the BFT/blockchain system. Nonetheless, at the end of the section we present some results in which malicious servers provide corrupted shares during the protocol execution.

In all experiments, the replicas use up to eight threads (half of the machine resources) for verifying and building commitments during recovery and resharing.

B. Normal Operation: Updates and Reads

Our first set of experiments aims to quantify the performance cost of having confidentiality on a KV store built on top of COBRA during normal operation. Table II presents the update throughput of this service considering update values of 1kB and groups of 4, 7, and 10 replicas. The table shows values for the same service without confidentiality (i.e., the KV store on top of BFT-SMaRt) as a reference, our VSSR implementation (with faster, linear commitments), and COBRA implementations using linear [13] and constant [26] ("con" variant) commitment schemes. Additionally, we evaluate COBRA with no share verification on the replicas (* variants), which does not satisfy linearizability for malicious clients operations [12], to highlight the overhead of such computation. It is worth to remark that the update protocol for MPSS is exactly the same we use in COBRA, so the latter results also serve for MPSS.

Unsurprisingly, BFT-SMaRt (without confidentiality) achieves the best throughput among the compared systems, with more than 15k updates processed per second with four replicas. COBRA without share verification reaches 67% (linear) and 88% (constant) of these values, being the variant that uses constant commitments more performant due to its smaller messages. When share verification is enabled, COBRA throughput drops significantly, especially when constant commitments are used.[6] Nonetheless, COBRA still achieves a throughput 3× better than VSSR (the state of the art). This difference comes from two factors: (1) VSSR updates are bigger due to the use of several shares, and, even more importantly, (2) it requires the verification of five shares (instead of one) on each update.

TABLE II
KV STORE THROUGHPUT (OPERATIONS/SEC) FOR DIFFERENT SYSTEMS CONSIDERING 4, 7, AND 10 REPLICAS AND UPDATES OF 1KB.

System	*Limitation*	$n=4$	$n=7$	$n=10$
BFT-SMaRt	No confidentiality	15353	10604	8028
COBRA*	No linearizability	10365	6311	4071
COBRA con*		13519	9458	7727
COBRA	None	2450	2114	1761
COBRA con		263	262	261
VSSR	No reconfiguration	776	591	481

As the number of replicas increases (i.e., the system tolerates more faults), the difference of throughput between BFT-SMaRt and COBRA decreases, and the throughput of COBRA constant remains approximately the same. This is due to the network becoming a bottleneck during consensus, and the cost of verifying shares in the constant commitment scheme being $O(1)$ (instead of $O(t)$, as in a linear commitment scheme).

The read throughput (not shown) varies between 44k to 39k in all systems with $n = 4$, with little degradation in bigger groups. This behavior comes from three factors: (1) no significant processing is done by replicas during reads, (2) read requests have the same size in all systems, and (3) only the reply sizes (e.g., due to the commitments) are different.

C. Recovery

Our second set of experiments measures the latency for recovering a replica with a non-trivial state containing 100k shares, considering systems with 4, 7, and 10 replicas. Table III presents the results for COBRA, VSSR, and BFT-SMaRt, factored in several columns: *Pol. Gen.* - the distributed polynomial generation (which includes the Byzantine consensus), *Blind.* - the share blinding, *Recon.* - the reconstruction of share and commitment, and *Other* - the data transfer and other computations (standard cryptography, serialization, etc).

The overall results show that the polynomial generation dominates the recovery latency of COBRA (accounting for up to 82% of total time). VSSR does not require this since it pre-generates the helper polynomials during updates, which makes its normal operation much slower. COBRA recovery is an order of magnitude slower than the BFT-SMaRt state transfer (which does not run COBRA DPSS), and 30% slower than VSSR in a group of 4 replicas. However, COBRA beats VSSR in bigger groups ($n = 7$ and $n = 10$) for two reasons. First, VSSR uses distributed pseudo-random functions that require expensive cryptographic computations [7]. Second, the

[6] Due to the lack of space, we report results using only linear commitments in the recovery and reshare experiments. Nonetheless, our results show that the use of constant commitments makes recovery (resp. reshare) 8× (resp. 12×) slower. This huge disparity between the two schemes was also observed in previous works [7], [15].

TABLE III
RECOVERY LATENCY (IN SECONDS) FOR A KV STORE REPLICA STATE WITH 100K ENTRIES (ONE SECRET/ENTRY) WITH 4, 7, AND 10 REPLICAS.

n	*Protocol*	*Pol. Gen.*	*Blind.*	*Recon.*	*Other*	*Total*
4	BFT-SMaRt	-	-	-	5.7	5.7
	VSSR	-	6.9	100.1	5.1	113.1
	COBRA	131.7	2.2	1.1	25.5	160.5
7	BFT-SMaRt	-	-	-	10.3	10.3
	VSSR	-	8.4	160.6	9.9	178.9
	COBRA	141.2	5.0	1.8	21.2	169.2
10	BFT-SMaRt	-	-	-	12.3	12.3
	VSSR	-	9.3	231.3	12.9	253.5
	COBRA	136.3	4.6	2.5	25.7	169.1

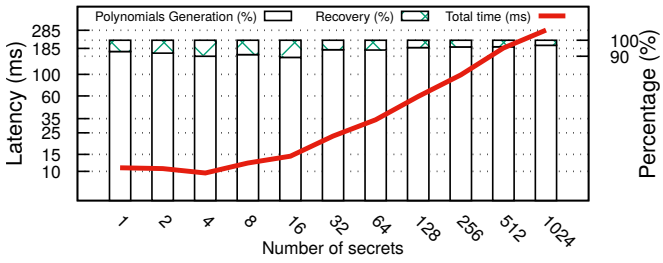

Fig. 4. COBRA recovery latency for 4 replicas ($t = 1$) and up to 1k secrets.

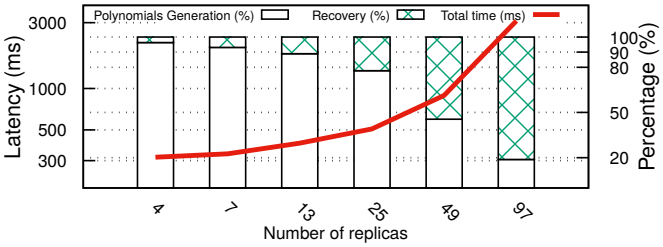

Fig. 5. COBRA recovery lat. for 1k secrets and up to 97 replicas ($t = 32$).

dominating factor in COBRA during recovery (i.e., polynomial generation) is mitigated using the Vandermonde matrix (see §VI-D). Therefore, as the group increases (i.e., tolerates more faults), the cost of running the VSSR recovery protocol increases much more than COBRA's.

To better understand the factors that impact COBRA, we performed micro benchmarks considering a variable number of secrets and replicas. Fig. 4 shows that the cost of recovering shares increases linearly with the number of shares (log scale used on both axes) and that the distributed polynomial generation accounts for about 90% of the latency of recovery.

The results of Fig. 5 show that, as n increases, the dominance of the recovery phase (Blind. plus Recon. on Table III) increases. With $n = 4$, recovering 1024 shares require 0.3s, being only 5% of this due to share and commitment recovery. This percentage goes to 80% when $n = 97$, with the recovery requiring more than 2.5s. This happens because the cost of polynomial generation is amortized by the use of the Vandermonde matrix optimization, which allows the generation of $t + 1$ polynomials on each execution of GeneratePolynomial. Naturally, as t increases, the weight of this protocol on the overall cost of recovery decreases. Recall that this phase also includes the execution of the PBFT-like Byzantine consensus protocol (see Fig. 2). However, in our experiments we noticed the contribution of this protocol to the overall latency is modest, when compared with the computational cost of VSS.

TABLE IV
RESHARE LATENCY (IN SECONDS) FOR A KV STORE REPLICA STATE WITH 100K ENTRIES (ONE SECRET/ENTRY) WITH 4, 7, AND 10 REPLICAS.

n	*Protocol*	*Pol. Gen.*	*Blind.*	*Recon.*	*Other*	*Total*
4	MPSS	550.7	17.4	65.4	12.3	645.8
	COBRA	283.3	2.1	15.8	18.1	319.3
7	MPSS	1721.1	27.9	91.7	21.9	1862.6
	COBRA	414.8	3.1	25.4	28.7	472.0
10	MPSS	3627.1	47.7	121.6	48.2	3844.5
	COBRA	665.1	5.0	32.0	41.7	743.8

D. Reshare

Our third set of experiments considers the latency for resharing the private state of all replicas in a group, without reconfiguring the replica set. We note that changing the system composition in this experiment will have negligible effect on the performance of the reshare protocol, since the executed steps would be exactly the same.

Table IV shows the latency of COBRA's resharing protocol for a private state of 100k secrets and groups of up to ten replicas. The columns have the same meaning as in Table III, since the high-level steps of the protocols are similar (see Fig. 3). Since VSSR does not support resharing, in these experiments we compare COBRA with MPSS [23].

Three observations can be made about these results. First, reshare is at least $2\times$ more costly than recovery, and much more affected by an increase on group size, as it does not use the Vandermonde matrix optimization. Second, the polynomial generation cost is the dominating factor followed by the shares and commitments reconstruction. Third, MPSS's reshare is $2 - 5\times$ slower than COBRA's (using the same linear commitments), with MPSS latency growing much faster than COBRA's as n increases. This happens because, although both protocols require a single consensus execution per secret reshare, MPSS generates $n + 1$ helper polynomials and their commitments, while COBRA uses only two polynomials and an optimized share verification scheme (see §VI-D).

As done for recovery, Figs. 6 and 7 present micro benchmark results for the COBRA reshare protocol with a variable number of secrets and replicas, respectively. The overall latency in both experiments increases linearly with the studied variables and the polynomial generation protocol accounts for 84% to 94% of the observed latency.

E. Impact of Malicious Servers

Our last set of experiments consider the impact of malicious servers in COBRA. In particular, we evaluate the impact of up to 3 malicious servers distributing invalid proposals (the main new attack vector introduced in COBRA) when running Reshare in a system with 10 replicas. We selected this protocol because it uses both Recover and GeneratePolynomial. Fig. 8

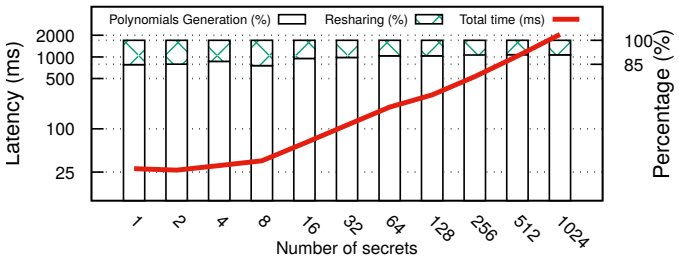

Fig. 6. COBRA reshare latency for 4 replicas ($t = 1$) and up to 1k secrets.

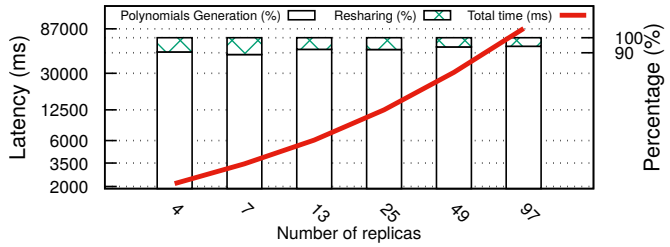

Fig. 7. COBRA reshare latency for 1k secrets and up to 97 replicas ($t = 32$).

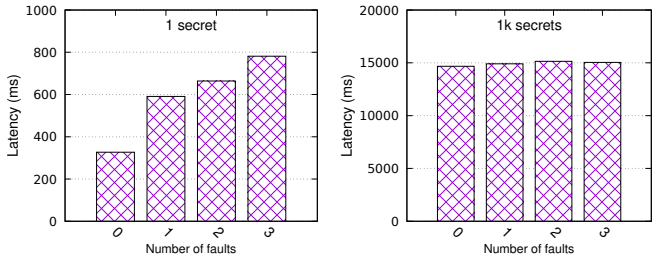

Fig. 8. COBRA reshare latency with failures in a system with 10 replicas.

presents latency results for a state containing 1 (left) and 1k (right) secrets.

With just 1 secret, the overhead of detecting and removing the malicious replica(s) and re-executing GeneratePolynomial unsurprisingly increases the latency of the protocol up to 58.2%. However, when running the protocol for 1k secrets, the observed overhead is almost negligible (2.4%). This happens because when faulty replicas provide corrupted shares, the performance of the protocols is degraded just for that execution, since COBRA detects and isolates the malicious nodes. Consequently, the impact of such invalid executions are diluted through the multiple executions required for refreshing 1k shares.

IX. Conclusion

This paper presents COBRA, a new protocol stack for DPSS that allows implementing confidentiality in practical BFT SMR systems. Compared with competing alternatives, COBRA exhibits minimal storage overhead and the best asymptotic communication complexity (when instantiated with optimal building blocks). Our experimental evaluation using BFT-SMaRt [24] shows that COBRA can recover (resp. reshare) a state with 100 000 secrets in a ten-replicas group 33% (resp. 5×) faster than VSSR [7] (resp. MPSS [23]).

Acknowledgment

We thank the IEEE S&P anonymous reviewers, Manuel Barbosa, and Bernardo Portela for their constructive comments to improve the paper. This work was supported by FCT through a PhD scholarship (2020.04412.BD), the ThreatAdapt project (FCT-FNR/0002/2018), the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020), the European Commission through the VEDLIoT project (H2020 957197), and CNPq (Brazil) through project number 420092/2018-8.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, 1990.

[2] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. on Programing Languages and Systems*, vol. 4, no. 3, 1982.

[3] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation – OSDI'99*, 1999.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[5] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Proc. of the Int. Workshop on Open Problems in Network Security*, 2015.

[6] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From Byzantine replication to blockchain: Consensus is only the beginning," in *Proc. of the 50th IEEE/IFIP Int. Conference on Dependable Systems and Networks – DSN'20*, 2020.

[7] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer, "Efficient verifiable secret sharing with share recovery in bft protocols," in *Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security – CCS'19*, 2019.

[8] S. Duan and H. Zhang, "Practical state machine replication with confidentiality," in *Proc. of the 35th IEEE Symp. on Reliable Distributed Systems – SRDS'16*, 2016.

[9] M. Khan and A. Babay, "Toward intrusion tolerance as a service: Confidentiality in partially cloud-based BFT systems," in *Proc. of the 51th IEEE/IFIP Int. Conference on Dependable Systems and Networks – DSN'21*, 2021.

[10] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, 1979.

[11] G. R. Blakley, "Safeguarding cryptographic keys," in *Proc. of the 1979 AFIPS National Computer Conference*, 1979.

[12] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "DepSpace: A Byzantine fault-tolerant coordination service," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[13] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *Proc. of the 28th IEEE Symp. on Foundations of Computer Science – FOCS'87*, 1987.

[14] E. Kokoris-Kogias, E. C. Alp, S. D. Siby, N. Gailly, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, "CALYPSO: Auditable sharing of private data over blockchains," Cryptology ePrint Archive, Report 2018/209, 2018.

[15] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song, "CHURP: Dynamic-committee proactive secret sharing," in *Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security – CCS'19*, 2019.

[16] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song, "Storing and retrieving secrets on a blockchain," Cryptology ePrint Archive, Report 2020/504, 2020.

[17] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks (extended abstract)," in *Proc. of the 10th ACM Symp. on Principles of Distributed Computing – PODC'91*, 1991.

[18] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO'95*, 1995.

[19] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Trans. on Computer Systems*, vol. 20, no. 4, 2002.

[20] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Trans. on Parallel and Distributed Systems*, vol. 21, no. 4, 2010.

[21] M. Garcia, A. Bessani, and N. Neves, "Lazarus: Automatic management of diversity in BFT systems," in *Proc. of the 20th Int. Middleware Conference – Middleware'19*, 2019.

[22] L. Zhou, F. B. Schneider, and R. Van Renesse, "APSS: Proactive secret sharing in asynchronous systems," *ACM Trans. on Information and System Security*, vol. 8, no. 3, 2005.

[23] D. Schultz, B. Liskov, and M. Liskov, "MPSS: Mobile proactive secret sharing," *ACM Trans. on Information and System Security*, vol. 13, no. 4, 2010.

[24] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *Proc. of the 44th IEEE/IFIP Int. Conference on Dependable Systems and Networks – DSN'14*, 2014.

[25] "COBRA source code," https://github.com/bft-smart/cobra, 2022.

[26] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT'10*, 2010.

[27] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync HotStuff: Simple and practical synchronous state machine replication," in *Proc. of the 41st IEEE Symp. on Security and Privacy – SP'20*, 2020.

[28] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. of the 2019 ACM Symp. on Principles of Distributed Computing – PODC'19*, 2019.

[29] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *Proc. of the 2002 ACM SIGSAC Conference on Computer and Communications Security – CCS'02*, 2002.

[30] Y. Desmedt and S. Jajodia, "Redistributing secret shares to new access structures and its applications," George Mason University, Tech. Rep. ISSE TR-97-01, 1997.

[31] T. M. Wong, C. Wang, and J. M. Wing, "Verifiable secret redistribution for archive systems," in *Proc. of the 1st Int. IEEE Security in Storage Workshop*, 2002.

[32] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky, "Communication-optimal proactive secret sharing for dynamic groups," in *Proc. of the 2015 Int. Conference on Applied Cryptography and Network Security – ACNS'15*, 2015.

[33] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO'91*, 1991.

[34] B. Schoenmakers, "A simple publicly verifiable secret sharing scheme and its application to electronic voting," in *CRYPTO'99*, 1999.

[35] L. Harn and C. Lin, "Detection and identification of cheaters in (t, n) secret sharing scheme," *Designs, Codes and Cryptography*, vol. 52, no. 1, 2009.

[36] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. G. Gueta, and S. Devadas, "Towards scalable threshold cryptosystems," in *Proc. of the 2020 IEEE Symp. on Security and Privacy – SP'20*, 2020.

[37] V. Nikov and S. Nikova, "On proactive secret sharing schemes," in *Proc. of the Int. Workshop on Selected Areas in Cryptography*, 2004.

[38] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO'01*, 2001.

[39] C. Cachin, "Yet another visit to Paxos," IBM Research Zurich, Tech. Rep. RZ 3754, 2009.

[40] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *Proc. of the 2020 ACM SIGSAC Conference on Computer and Communications Security – CCS'20*, 2020.

[41] T.-H. H. Chan, R. Pass, and E. Shi, "Pala: A simple partially synchronous blockchain," Cryptology ePrint Archive, Report 2018/981, 2018.

[42] J. S. Fraga and D. Powell, "A fault- and intrusion-tolerant file system," in *Proc. of the 3rd IFIP Int. Conference on Computer Security – SEC'85*, 1985.

[43] S. Lakshmanan, M. Ahamad, and H. Venkateswaran, "Responsive security for stored data," *IEEE Trans. on Parallel and Distributed Systems*, vol. 14, 2003.

[44] M. A. Marsh and F. B. Schneider, "CODEX: a robust and secure secret distribution system," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, 2004.

[45] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, 1998.

[46] R. Padilha and F. Pedone, "Belisarius: BFT storage with confidentiality," in *Proc. of the 10th IEEE Int. Symp. on Network Computing and Applications – NCA'11*, 2011.

[47] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proc. of the 19th ACM Symp. on Theory of Computing – STOC'87*, 1987.

[48] Y. Lindell, "Secure multiparty computation," *Communications of the ACM*, vol. 64, no. 1, 2021.

[49] A. Barak, M. Hirt, L. Koskas, and Y. Lindell, "An end-to-end system for large scale P2P MPC-as-a-Service and low-bandwidth MPC for weak participants," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security – CCS'18*, 2018.

[50] A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk, "Fluid mpc: Secure multiparty computation with dynamic participants," in *CRYPTO'21*, 2021.

[51] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, 1988.

[52] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security – CCS'17*, 2017.

[53] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *Proc. of the 2013 Int. Conference on Applied Cryptography and Network Security – ACNS'13*, 2013.

[54] D. Beaver and S. Haber, "Cryptographic protocols provably secure against dynamic adversaries," in *Proc. of the Workshop on the Theory and Application of of Cryptographic Techniques*, 1992.

[55] M. Hirt, C.-D. Liu-Zhang, and U. Maurer, "Adaptive security of multi-party protocols, revisited," in *Proc. of the Theory of Cryptography Conference – TCC'21*, 2021.

[56] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 3, 1990.

[57] M. Herlihy, "Wait-free synchronization," *ACM Trans. on Programming Languages and Systems*, vol. 13, no. 1, 1991.

[58] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Proc. of the 9th European Dependable Computing Conference – EDCC'12*, 2012.

[59] T. Crain, C. Natoli, and V. Gramoli, "Red belly: A secure, fair and scalable open blockchain," in *Proc. of the 42nd IEEE Symp. on Security and Privacy – SP'21*, 2021.

[60] A. Bessani, M. Santos, J. a. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. of the USENIX Annual Technical Conference – ATC'13*, 2013.

[61] M. Eischer, M. Büttner, and T. Distler, "Deterministic fuzzy checkpoints," in *Proc. of the 38th Int. Symp. on Reliable Distributed Systems – SRDS'19*, 2019.

[62] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proc. of the Very Large Data Base Endowment – VLDB'14*, vol. 8, no. 3, 2014.

[63] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *CRYPTO'07*, 2007.

[64] W. G. Horner, "A new method of solving numerical equations of all orders, by continuous approximation," *Philosophical Trans. of the Royal Society of London*, vol. 109, 1819.

[65] Relic Deveplopment Team, "The relic toolkit crypto library (relic)," https://github.com/relic-toolkit/relic, 2021.

[66] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: a scalable decentralized trust infrastructure for blockchains," in *Proc. of the 49th IEEE/IFIP Int. Conference on Dependable Systems and Networks – DSN'19*, 2019.

[67] J. Behl, T. Distler, and R. Kapitza, "Hybrids on steroids: SGX-based high performance BFT," in *Proc. of the 12th ACM SIGOPS/EuroSys European Conference on Computer Systems – EuroSys'17*, 2017.

APPENDIX

A. Modified Distributed Polynomial Generation

In the following we present the GeneratePolynomial$*$ protocol used in COBRA's Reshare protocol (§V-C). The main difference between this protocol and the one presented in §V-C is that two polynomials Q and Q' are generated at once, i.e., servers in $\mathcal{V}_{old}$ obtain shares from Q while servers in $\mathcal{V}_{new}$ obtain shares from Q', both with the same random independent term.

PROTOCOL GeneratePolynomial$*(\mathcal{V}_{old}, \mathcal{V}_{new})$

Server $r_i \in \mathcal{V}_{old}$ with inputs $(\mathcal{V}_{old}, \mathcal{V}_{new})$:

S1. Invokes $\langle\{s_{i,1}, ..., s_{i,\mathcal{V}_{old}.n}\}, c_i\rangle = \mathsf{Share}(\mathcal{V}_{old}, 0, q_i)$ and $\langle\{s'_{i,1}, ..., s'_{i,\mathcal{V}_{new}.n}\}, c'_i\rangle = \mathsf{Share}(\mathcal{V}_{new}, 0, q_i)$, being q_i a random number.

S2. Sends $\langle \mathsf{PROPOSAL}, E_1(s_{i,1})...E_{\mathcal{V}_{old}.n}(s_{i,\mathcal{V}_{old}.n}), c_i, c'_i\rangle_{\sigma_i}$ to $\mathcal{V}_{old}$ and $\langle \mathsf{PROPOSAL}, E_1(s'_{i,1})...E_{\mathcal{V}_{new}.n}(s'_{i,\mathcal{V}_{new}.n}), c_i, c'_i\rangle_{\sigma_i}$ to $\mathcal{V}_{new}$.

Server $r_i \in \mathcal{V}_{old} \cup \mathcal{V}_{new}$ with inputs $(\mathcal{V}_{old}, \mathcal{V}_{new})$:

S3. Collects PROPOSAL messages and runs Consensus:
   a) Let $r_l \in \mathcal{V}_{old}$ be an elected consensus leader. r_l builds a set $S = \{\langle j, c_j, c'_j\rangle : r_j \in \mathcal{V}_{old}\}$ using $\mathcal{V}_{old}.t + 1$ received *valid proposals* (a proposal from r_j is valid for r_l iff $\langle \mathsf{PROPOSAL}, ...E_l(s_{j,l})..., c_j, c'_j\rangle_{\sigma_j}$ is correctly signed and both $\mathsf{Verify}(l, s_{j,l}, c_j)$ and $\mathsf{SameIT}(c_j, c'_j)$ returns *true*) and proposes this set to $\mathcal{V}_{old} \cup \mathcal{V}_{new}$.
   b) A correct replica accepts the set S proposed by r_l if it received valid proposals from all r_j such that $\langle j, *, *\rangle \in S$ (the $\mathcal{P}$-Validity predicate).
   c) During the protocol execution, a quorum is formed by waiting $\mathcal{V}_{old}.n - \mathcal{V}_{old}.t$ from $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.n - \mathcal{V}_{new}.t$ from $\mathcal{V}_{new}$.

S4. Waits for Consensus to output S. For each tuple $\langle j, c_j, c'_j\rangle \in S$, r_i asks for r_j's proposal $\langle \mathsf{PROPOSAL}, ...E_i(s*_{j,i})..., c_j, c'_j\rangle_{\sigma_j}$ in $\mathcal{V}_{old} \cup \mathcal{V}_{new}$ if it does not have it.

S5. Outputs generated shares:
   a) If $r_i \in \mathcal{V}_{old}$: decrypts all $E_i(s_{j,i})$, computes $s_i^q = \sum_{\langle j,*,*\rangle \in S} s_{j,i}$ and $c_q = \bigotimes_{\langle j,*,*\rangle \in S} c_j$, and outputs $\langle S, s_i^q, c_q\rangle$.
   b) If $r_i \in \mathcal{V}_{new}$: decrypts all $E_i(s'_{j,i})$, computes $s_i^{q'} = \sum_{\langle j,*,*\rangle \in S} s'_{j,i}$ and $c_{q'} = \bigotimes_{\langle j,*,*\rangle \in S} c'_j$, and outputs $\langle S, s_i^{q'}, c_{q'}\rangle$.

The protocol follows the same structure as its single-polynomial counterpart, with three phases and five steps. However, there are subtle but important differences. Steps S1 and S2 are only executed by servers in $\mathcal{V}_{old}$. In S1, instead of generating a polynomial encoding $(x, 0)$ for a given input x, each server $r_i \in \mathcal{V}_{old}$ generates two different random polynomials Q_i and Q'_i encoding the same point $(0, q_i)$, being q_i a random value. In S2, servers in $\mathcal{V}_{old}$ exchange proposals with encrypted shares of Q_* and send proposals with encrypted shares of Q'_* to $\mathcal{V}_{new}$. During consensus (Step S3), servers, besides verifying the validity of their shares, use the predicate $\mathsf{SameIT}(c_j, c'_j)$ to check if both polynomials encode the same independent term, i.e., $Q_i(0) = Q'_i(0)$. This is done by comparing the (hidden) evaluation of both polynomials at $x = 0$ using c_j and c'_j.[7] Finally, when running Consensus, instead of waiting for messages from a quorum of $\mathcal{V}.n - \mathcal{V}.t$ processes before proceeding to the next phase, each server waits for composite quorums containing $\mathcal{V}_{old}.n - \mathcal{V}_{old}.t$ servers from $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.n - \mathcal{V}_{new}.t$ servers from $\mathcal{V}_{new}$. In the end, the protocol ensures that at least $\mathcal{V}_{old}.t+1$ correct servers from $\mathcal{V}_{old}$ can reconstruct Q (Step 5a) and $\mathcal{V}_{new}.t+1$ correct servers from $\mathcal{V}_{new}$ obtain valid shares of Q' (Step 5b).

B. COBRA DPSS Security

Formally, the semantic security of a DPSS scheme is defined w.r.t. an ideal functionality $\mathcal{F}^{DPSS}$, which is built on top of ideal functionalities $\mathcal{F}^{VSS}$ and $\mathcal{F}^{Pol}$. $\mathcal{F}^{DPSS}$ describes the behavior of a secure DPSS scheme in an idealized world and specifies all information that each of its operations is allowed to reveal, which is captured by a leakage function $\mathcal{L}^{DPSS}$. Moreover, $\mathcal{F}^{VSS}$ describes the behavior of a secure VSS scheme with leakage $\mathcal{L}^{VSS}$ and $\mathcal{F}^{Pol}$ of a polynomial generation scheme with leakage $\mathcal{L}^{Pol}$. Then, the onus of the security proof consists in showing that in the real world $\mathcal{F}^{DPSS}$, $\mathcal{F}^{VSS}$, and $\mathcal{F}^{Pol}$ can be securely replaced by concrete implementations $\prod^{DPSS}$, $\prod^{VSS}$, and $\prod^{Pol}$, respectively, iff they exhibit the same leakage functions $\mathcal{L}^{DPSS}$, $\mathcal{L}^{VSS}$, and $\mathcal{L}^{Pol}$ and reveal nothing else to an adversary.

In more detail, we describe ideal functionalities $\mathcal{F}^{DPSS}$, $\mathcal{F}^{VSS}$, and $\mathcal{F}^{Pol}$, each with n party interfaces, an adversary interface A, and a free interface W. The party interfaces allow each party $i \in [n]$ to access the ideal functionalities and input their respective operations, namely Share, Verify, Reconstruct, Recover, and Reshare for $\mathcal{F}^{DPSS}$, Share, Verify, and Reconstruct for $\mathcal{F}^{VSS}$, and GeneratePolynomial plus GeneratePolynomial$*$ for $\mathcal{F}^{Pol}$. In the real world these actions map to the invocation of protocols $\prod^{DPSS} = \{\mathsf{Share}, \mathsf{Verify}, \mathsf{Reconstruct}, \mathsf{Recover}, \mathsf{Reshare}\}$, $\prod^{VSS} = \{\mathsf{Share}, \mathsf{Verify}, \mathsf{Reconstruct}\}$, and $\prod^{Pol} = \{\mathsf{GeneratePolynomial}, \mathsf{GeneratePolynomial}*\}$ with the same name. Moreover, interface A allows the adversary to leak the internal state and control the inputs and outputs of corrupted parties, while W allows the ideal functionalities to keep track of the set of corrupted parties.

Given their ideal functionalities, $\mathcal{L}^{DPSS}$, $\mathcal{L}^{VSS}$, and $\mathcal{L}^{Pol}$ will only reveal how many operations have been carried out and their types, plus the sizes (if they are private) or values (if they are public) of inputs and outputs for these operations. Additionally, $\mathcal{L}^{Pol}$ will reveal the generated polynomial when running GeneratePolynomial, as it only requires the generated polynomial to be random.

Following the CC-adaptive framework of Hirt et al. [55], the proof must show that in the ideal world, and for each of the three functionalities described above, there exists a set of simulators $\mathcal{S}$, with one simulator for each possible set of correct replicas, that can model all interactions between the

[7] In the constant commitment scheme [26], servers have to compute additional witnesses for $(0, q_i)$ to check this equality.

adversary and each functionality, and that these interactions are computationally indistinguishable from its interactions with the respective implementations $\prod^*$ in the real world. Adaptive corruptions are modeled in the CC-adaptive framework by having every simulator $S_X \in \mathcal{S}$ attached to the adversary interface of each functionality. Further, when a party in X is corrupted, S_X halts. By describing all these simulators, which are essentially the same, albeit with different corrupted parties, we model all possible adaptive corruptions without incurring in the technical difficulties of UC-adaptive model.

Next we present a proof sketch for the security of COBRA DPSS ($\prod^{DPSS}$) and of our distributed polynomial generation scheme ($\prod^{Pol}$). Since we use VSS as a building block based on well-known schemes [13], [26], we defer to their papers for security proofs of $\prod^{VSS}$. Our sketches are centered around the Secrecy, Integrity, and Termination properties as stated in Definition 5.2 (§V-B), which informally model the guarantees provided by ideal functionalities $\mathcal{F}^{DPSS}$ and $\mathcal{F}^{Pol}$ and their leakage functions $\mathcal{L}^{DPSS}$ and $\mathcal{L}^{Pol}$.

1) Secrecy: As described in §V, COBRA DPSS is composed by the Recover and Reshare protocols, which are assisted by the GeneratePolynomial and GeneratePolynomial$*$ protocols, respectively.

Let $Corrupt(\mathcal{V})$ be the set of up to $\mathcal{V}.t$ faulty servers controlled by adversary $\mathcal{A}$ in $\mathcal{V}$. For each protocol, and besides its public inputs, $\mathcal{A}$ has access to the following information:

GeneratePolynomial:

- $\forall r_i \in Corrupt(\mathcal{V})$, shares $s_{*,i}$ and commitment c_* of all polynomials P_* proposed by correct servers;
- The set $S = \{\langle j, c_j\rangle : r_j \in \mathcal{V}\}$ decided by $\mathcal{V}$;
- $\forall r_i \in Corrupt(\mathcal{V})$, share s_i and commitment c of the final polynomial P.

GeneratePolynomial$*$:

- $\forall r_i \in Corrupt(\mathcal{V}_{old})$, shares $s_{*,i}$ and commitment c_* of all polynomials P_* proposed by correct servers in $\mathcal{V}_{old}$;
- $\forall r_i \in Corrupt(\mathcal{V}_{new})$, shares $s'_{*,i}$ and commitment c'_* of all polynomials P'_* proposed by correct servers in $\mathcal{V}_{new}$;
- The set $S = \{\langle j, c_j, c'_j\rangle : r_j \in \mathcal{V}_{old}\}$ decided by $\mathcal{V}_{new} \cup \mathcal{V}_{old}$;
- $\forall r_i \in Corrupt(\mathcal{V}_{old})$, share s_i^q and commitment c^q of the final polynomial Q;
- $\forall r_i \in Corrupt(\mathcal{V}_{new})$, share $s_i^{q'}$ and commitment $c^{q'}$ of the final polynomial Q'.

Recover:

- $\forall r_i \in Corrupt(\mathcal{V})$, $\langle S, s_i^r, c^r\rangle$ resulting from the invocation of GeneratePolynomial;
- $\forall r_i \in Corrupt(\mathcal{V})$, accusation proposals $\langle \mathsf{ACC}, j, prop_j, s_{j,l}\rangle$ sent by r_k;
- If recovering server r_k is corrupted, blinded polynomial $B(.) = P(.) + R(.)$, recovered share s_k, commitments c and c^r, and accusation proposals $\langle \mathsf{ACC}, j, prop_j, s_{j,i}\rangle$ sent by servers $r_i \in \mathcal{V}$;

Reshare:

- $\forall r_i \in Corrupt(\mathcal{V}_{old})$, $\langle S, s_i^q, c^q\rangle$ resulting from the invocation of GeneratePolynomial$*$;
- $\forall r_i \in Corrupt(\mathcal{V}_{new})$, $\langle S, s_j^{q'}, c^{q'}\rangle$ resulting from the invocation of GeneratePolynomial*;
- The blinded polynomial $B(.) = P(.) + Q(.)$ and commitment c^b;
- $\forall r_j \in \mathcal{V}_{new}$ that invokes Recover, and $\forall r_i \in Corrupt(\mathcal{V}_{new})$, the shares and commitments $\langle S_j, s_i^{r_j}, c^r\rangle$, the blinded polynomials B_j if r_j is also corrupted, and the accusations $\langle \mathsf{ACC}, k, prop_k, s_{k,l}\rangle$;
- $\forall r_i \in Corrupt(\mathcal{V}_{new})$, share s'_j and commitment c'.

To prove secrecy we have the following lemmata.

LEMMA A.1: Given a VSS scheme with Hiding and if $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers in $\mathcal{V}$, then polynomial P generated in GeneratePolynomial is randomly generated.

Proof: By the Hiding property of the VSS scheme employed, the local polynomials generated by correct servers in $\mathcal{V}$ will be randomly generated. Since leader r_l defines S by selecting at least $\mathcal{V}.t + 1$ of such polynomials sent in valid proposals, there will always be at least one polynomial in S whose generation can not be influenced by $\mathcal{A}$. Even if r_l is controlled by $\mathcal{A}$, it will still not be able to select $\mathcal{V}.t + 1$ corrupt proposals for S, since there are at most $\mathcal{V}.t$ corrupted servers in $\mathcal{V}$ and correct servers only accept S if they received valid polynomial proposals from the servers indicated in S. By the Hiding property of the underlying commitment scheme, commitments c_* of locally generated polynomials P_*, and hence commitment c of final polynomial P, also reveal no additional information to $\mathcal{A}$. Consequently, $\mathcal{A}$ can not influence the generation of P and, even though it ends up with $\mathcal{V}.t + 1$ shares of P ($P(x) = 0$ is a public parameter), P will still be randomly generated. ∎

LEMMA A.2: Given a VSS scheme with Hiding and if $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$, then polynomials Q and Q' generated in GeneratePolynomial$*$ are randomly generated and secret.

Proof: This proof is similar to the proof of Lemma A.1. Polynomials Q and Q' will be randomly generated since $\mathcal{A}$ can corrupt at most $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$. Nonetheless, in this proof we must also show that besides randomly generating polynomials Q and Q', GeneratePolynomial$*$ also ensures their secrecy.

Secrecy is ensured by making every honest server r_i in $\mathcal{V}_{old}$ invoke Share with a secret and randomly generated value q_i and by ensuring that $\mathcal{A}$ never learns more than $\mathcal{V}_{old}.t$ shares of polynomial Q and $\mathcal{V}_{new}.t$ shares of polynomial Q'. Given this, and considering the Hiding property of the VSS scheme employed and that correct replicas erase secret values q_i after invoking Share, polynomials Q and Q' will be randomly generated and secret. Moreover, as in Lemma A.1, commitments c^q and $c^{q'}$ also reveal no additional information to $\mathcal{A}$ due to the Hiding property of the commitment scheme. ∎

LEMMA A.3: If $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers in $\mathcal{V}$ and GeneratePolynomial randomly generates polynomial R, then the information received by $\mathcal{A}$ in Recover is random and independent of secret s.

Proof: By Lemma A.1, the GeneratePolynomial protocol will generate a new random polynomial R and distribute its shares through $\mathcal{V}$. Two situations can occur then:

1) If $r_k \notin Corrupt(\mathcal{V})$, $\mathcal{A}$ can still reconstruct R using $t+1$ shares since it may control $\mathcal{V}.t$ servers and r_k's share is publicly known (point $(k,0)$). However, if it does not corrupt r_k, then it does not have access to blinded polynomial $B(.) = P(.) + R(.)$ and can not reconstruct P by calculating $P(.) = B(.) - R(.)$.
2) If $r_k \in Corrupt(\mathcal{V})$, $\mathcal{A}$ only has access to $\mathcal{V}.t-1$ shares of R and r_k's point $(k,0)$. Hence, it can not reconstruct R and can not calculate $P(.) = B(.) - R(.)$.

Recovering $s_k = B(k)$ also does not give $\mathcal{A}$ any additional advantage. Additionally, after recovering its share s_k, r_k discards polynomial B. Hence, even if r_k is later corrupted by $\mathcal{A}$, it will still not be able to reconstruct P. Moreover, accusation messages $\langle \mathsf{ACC}, j, prop_j, s_{j,i} \rangle$ reveal no additional information to $\mathcal{A}$, since these messages are only sent if shares $s_{j,i}$ are invalid according to the commitment sent in $prop_j$. Finally, by the Hiding property of the underlying commitment scheme, the commitments of all polynomials involved in Recover reveal no additional information to $\mathcal{A}$. ∎

LEMMA A.4: If (1) $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$, (2) GeneratePolynomial$*$ randomly generates secret polynomials Q to $\mathcal{V}_{old}$ and Q' to $\mathcal{V}_{new}$, and (3) Recover allows any server $r_j \in \mathcal{V}_{new}$ to obtain a valid share of Q', then the information received by $\mathcal{A}$ in Reshare is random and independent of secret s.

Proof: By Lemma A.2, protocol GeneratePolynomial$*$ will generate new random and secret polynomials Q and Q', distributing their shares through $\mathcal{V}_{old}$ and $\mathcal{V}_{new}$, respectively. Since $\mathcal{A}$ can corrupt at most $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$, it can not reconstruct Q and Q' from their shares. Hence, even knowing the blinded polynomial $B(.) = P(.) + Q(.)$, it can not recover P by calculating $P(.) = B(.) - Q(.)$.

Additionally, calculating $s'_j = B(0) - Q'(j)$ for every corrupted server r_j in $\mathcal{V}_{new}$ does not give $\mathcal{A}$ any additional advantage. By Lemma A.3, every $r_j \in \mathcal{V}_{new}$ that invokes Recover, reveals no additional information to $\mathcal{A}$. Finally, by the Hiding property of the underlying commitment scheme, the commitments of all polynomials involved in Reshare reveal no additional information to $\mathcal{A}$. ∎

By Lemmata A.3 and A.4, $\mathcal{A}$ does not learn any information about the secret in any view $\mathcal{V}$, which suffices to prove COBRA DPSS *Secrecy* (Definition 5.2).

2) Integrity: For proving integrity we have to show that the shares generated in Recover and Reshare can still be used to reconstruct the original secret. We start by proving auxiliary lemmata for GeneratePolynomial and GeneratePolynomial$*$.

LEMMA A.5: Given a VSS scheme with Binding, if $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers in $\mathcal{V}$, then GeneratePolynomial generates a random polynomial P with $P(x) = 0$ and at least $\mathcal{V}.t+1$ honest servers will be able to reconstruct it.

Proof: During GeneratePolynomial, each honest server in $\mathcal{V}$ executes Share to locally generate a random polynomial P_* with $P_*(x) = 0$ and commitment c_*. These polynomials are sent to servers in $\mathcal{V}$ and a Consensus leader r_l selects a set $S = \{\langle j, c_j \rangle : r_j \in \mathcal{V}\}$ identifying $\mathcal{V}.t+1$ valid proposals. A signed proposal from server r_l is considered *valid* by server r_i iff $\forall \langle j, c_j \rangle \in S$, both Verify$(x, 0, c_j)$ and Verify$(i, s_{j,i}, c_j)$ returns *true*. By the Binding property of the VSS scheme, and assuming the security of the commitment scheme and of the public-key signature scheme used, honest nodes can correctly validate shares received in proposals. By the $\mathcal{P}$-*Validity* property of Consensus, it is ensured that at least $\mathcal{V}.t+1$ honest servers accept the proposals in the decided set S, i.e., they are valid for them. Consequently, all r_i in this set of at least $\mathcal{V}.t+1$ honest servers generate their $s_i = P(i) = \sum_{\langle j,* \rangle \in S} s_{j,i}$ and commitment $c = \bigotimes_{\langle *, c_j \rangle \in S} c_j$. Since P is the sum of $\mathcal{V}.t+1$ polynomials P_* with $P_*(x) = 0$, $P(x) = 0$ (due to the proposals validity). Finally, these honest servers can provide their valid shares s_* to reconstruct P. ∎

LEMMA A.6: Given a VSS scheme with Binding, if $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ in $\mathcal{V}_{new}$, then the GeneratePolynomial$*$ protocol generates two random polynomials Q and Q', such that $Q(0) = Q'(0) = q$ (being q a random value), and at least $\mathcal{V}_{old}.t+1$ correct servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t+1$ correct servers in $\mathcal{V}_{new}$ will be able to reconstruct Q and Q', respectively.

Proof: This proof is similar to the proof of Lemma A.5, and hence omitted due to space constraints. ∎

LEMMA A.7: If $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers in $\mathcal{V}$, and GeneratePolynomial correctly generates a random polynomial R with $R(k) = 0$ that can be reconstructed by at least $\mathcal{V}.t+1$ correct servers, then, after Recover, recovering server r_k obtains a share $s_k = P(k)$ and commitment c.

Proof: Consider a server r_k recovering its share by sending a request to others servers in $\mathcal{V}$. In response, other correct servers execute GeneratePolynomial to generate R with $R(k) = 0$. By Lemma A.6, at least $\mathcal{V}.t+1$ correct servers will generate shares for R. These servers send blinded shares $B(i) = P(i) + R(i)$, c, and c^r to r_k. Server r_k eventually receives at least $\mathcal{V}.t+1$ of such messages that are valid and with the same c and c^r. Thus, and assuming the security of the employed commitment scheme, the $\mathcal{V}.t$ corrupted servers in $\mathcal{V}$ are not able to make r_k reconstruct an invalid polynomial B'. Consequently, r_k computes $s_k = B(k) = (P+R)(k) = P(k)$, which is verifiable by c. ∎

LEMMA A.8: If (1) $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and no more than $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$, (2) GeneratePolynomial$*$ randomly generates secret polynomials Q and Q' with $Q(0) = Q'(0)$ that can be reconstructed by at least $\mathcal{V}_{old}.t+1$ and $\mathcal{V}_{new}.t+1$ correct servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}$, respectively, and (3) Recover allows any server $r_j \in \mathcal{V}_{new}$ to obtain a valid share of Q', then the shares generated for honest servers in $\mathcal{V}_{new}$ preserve the same shared secret s.

Proof: By Lemma A.6, at least $\mathcal{V}_{old}.t+1$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t+1$ in $\mathcal{V}_{new}$ obtain valid shares of Q and

Q', respectively. Hence, at least $\mathcal{V}_{old}.t + 1$ correct servers in $\mathcal{V}_{old}$ generate blinded shares $s_i^b = s_i + Q(i)$ and send them to $\mathcal{V}_{new}$. Consequently, and assuming the security of the commitment scheme used, at least $\mathcal{V}_{new}.t + 1$ servers in $\mathcal{V}_{new}$ collect $\mathcal{V}_{old}.t + 1$ blinded shares that are valid, and reconstruct $z = B(0)$. Correct servers in $\mathcal{V}_{new}$ with invalid shares of Q' execute the Recover protocol, which, by Lemma A.7, recover their shares. Using z and $Q'(i)$, r_j computes $s'_j = z - Q'(j)$. Secret s remains intact since $s = P(0) = P(0) + Q(0) - Q'(0) = z - Q'(0)$. ■

By Lemmata A.7 and A.8, $\mathcal{A}$ can not prevent honest servers in any view $\mathcal{V}$ from computing their shares and reconstructing secret s, which suffices to prove COBRA DPSS *Integrity* (Definition 5.2).

3) Termination: For proving termination we have to show that $\mathcal{A}$ cannot prevent Recover and Reshare to terminate in correct servers. We start by proving auxiliary lemmata for GeneratePolynomial and GeneratePolynomial$*$.

LEMMA A.9: If $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers, then GeneratePolynomial terminates.

Proof: The system contains at least $\mathcal{V}.n - \mathcal{V}.t$ correct servers in $\mathcal{V}$ that eventually start the distributed polynomial generation protocol by sending valid proposals to $\mathcal{V}$. Consequently, it is ensured that all correct servers in $\mathcal{V}$ eventually receive at least $\mathcal{V}.n - \mathcal{V}.t$ of such proposals and some correct leader will be able to select $\mathcal{V}.t + 1$ valid proposals for it and define S to propose in Consensus. By the Byzantine consensus' *Termination* and $\mathcal{P}$*-Validity* properties, the decided set S was approved by at least $\mathcal{V}.t + 1$ correct servers. These servers will provide these proposals to any other correct server that did not received them. After that, shares and commitments are computed locally and the protocol terminates. ■

LEMMA A.10: If $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ in $\mathcal{V}_{new}$, then GeneratePolynomial$*$ terminates.

Proof: This proof is similar to the proof of Lemma A.9, and hence omitted due to space constraints. ■

LEMMA A.11: If $\mathcal{A}$ corrupts no more than $\mathcal{V}.t$ servers in $\mathcal{V}$, then Recover terminates.

Proof: Consider a server $r_k \in \mathcal{V}$ recovering its share by sending a request to others servers. Since the GeneratePolynomial protocol terminates (Lemma A.9) with at least $\mathcal{V}.t + 1$ correct servers in $\mathcal{V}$ obtaining valid shares of the generated polynomial R (Lemma A.5), if a correct server receives an invalid share of R from some server r_j, it accuses it of sending invalid proposals during a GeneratePolynomial execution. All correct servers that receive this accusation and validate its soundness, start to ignore r_j from now on. Moreover, a malicious recovering server r_k is not able to forge that a server r_l is accusing r_j because it does not have the decrypted share $s_{j,l}$ to prove it is invalid. Consequently, since (1) r_k restarts the protocol after each accusation, (2) each valid accusation causes a corrupt server to be ignored, and (3) there are at most $\mathcal{V}.t$ corrupt servers in $\mathcal{V}$, eventually all correct servers in $\mathcal{V}$ obtain valid shares of R. Since at least $\mathcal{V}.t+1$ correct servers have shares and commitments of P (the polynomial whose r_k is trying to recover its share), they will create blinded shares, and send this information to r_k, which calculates its share and commitment, terminating the protocol. ■

LEMMA A.12: If $\mathcal{A}$ corrupts no more than $\mathcal{V}_{old}.t$ servers in $\mathcal{V}_{old}$ and $\mathcal{V}_{new}.t$ servers in $\mathcal{V}_{new}$, then Reshare terminates.

Proof: By Lemma A.10, GeneratePolynomial$*$ terminates. By Lemma A.6, at least $\mathcal{V}_{old}.t + 1$ correct servers in $\mathcal{V}_{old}$ obtain valid shares of polynomial Q and send valid blinded shares to $\mathcal{V}_{new}$, terminating the protocol. The remaining servers in $\mathcal{V}_{old}$ with invalid shares of Q also terminate the protocol. By Lemma A.6, at least $\mathcal{V}_{new}.t + 1$ correct servers in $\mathcal{V}_{new}$ obtain valid shares of polynomial Q', and wait for the $\mathcal{V}_{old}.t + 1$ valid blinded shares from $\mathcal{V}_{old}$. These servers compute their renewed shares and terminate the protocol. The correct servers in $\mathcal{V}_{new}$ that receive invalid shares of Q' execute the Recover protocol to obtain valid shares of Q'. Since by Lemma A.11 this protocol terminates, these servers will also terminate Reshare. ■

By Lemmata A.11 and A.12, $\mathcal{A}$ can not prevent honest servers from completing the COBRA DPSS protocols, proving thus *Termination* (Definition 5.2).

C. Confidential BFT SMR Security

In this section we outline the main arguments for the security of COBRA BFT SMR. Recall that a confidential BFT SMR service is considered secure if it fulfills Safety, Liveness, and Secrecy (see §III-E). The first two properties follows directly from the underlying BFT SMR framework (which includes a Byzantine consensus protocol) and COBRA DPSS' Integrity and Termination properties. However, Secrecy, the main new property COBRA brings to BFT SMR, requires a bit more of discussion.

Recall that COBRA BFT SMR global state $S = \{D_1, ..., D_m\}$ is composed of m data entries, with each correct server r_i with a state $S_i = \langle C, P_i \rangle$ (see §III-C). Further, four protocols are supported: Update, Read, StateRecover, and Reconfigure. Besides public data, adversary $\mathcal{A}$ has access to the following information:

- All data entries $D_c \in S$ accessible to corrupted clients;
- If $Corrupt(\mathcal{V}) \neq \emptyset$, the service common state $C = \{\langle \overline{D}_j, D_j^e, c_j \rangle\}_{j \in [m]}$;
- $\forall r_i \in Corrupt(\mathcal{V})$, its private state $P_i = \{s_{i,j}\}_{j \in [m]}$;
- For invocations of Update and Read, no additional leakage is provided to $\mathcal{A}$.
- For invocations of StateRecover and Reconfigure, all the leakage from invoking COBRA DPSS Recover and Reshare, respectively.

Assuming that $\mathcal{A}$ can corrupt at most $\mathcal{V}.t$ servers in the current view $\mathcal{V}$, and from the security of COBRA DPSS, $\mathcal{A}$ cannot recover data entries from the private state of corrupted replicas. Additionally, from the Secrecy property of COBRA DPSS, the information leaked in StateRecover and Reconfigure also does not reveal any additional information to $\mathcal{A}$ for all stored data entries $D \in S$.