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 Brasília, 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× 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 re-share [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 re-share 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 resharin) 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 \( s_1, \ldots, s_n \) 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 \( s_1, \ldots, s_n \) as points of \( P \), i.e.,

\[
\hat{s}_i = P(i).
\]

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
scheme must be extended with commitments, implementing a Verifiable Secret Sharing (VSS) [13], [26], [33]–[36]. A (non-
interacting) 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
\[ P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_t x^t \]
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 $g^y = (g^{a_0})^x \times ... \times (g^{a_t})^x \times (g^{a_t})$ is true.

A Proactive Secret Sharing (PSS) scheme can protect the
secrecy of the data in the 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\(^\dagger\) 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
asynchronous 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 differ-
ent 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
(DPSS), or just DPSS for simplicity.

Table I summarizes some of the characteristics of existing
schemes and COBRA DPSS, including the fault-free commu-
nications complexity of share/combine and resharable 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-asynchronous
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 com-
plexity (which is optimal), but has a high communication
complexity on resharable 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 a also a polynomial, but a cubic resharable
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

### Table I

<table>
<thead>
<tr>
<th>PSS Scheme</th>
<th>Synchronous</th>
<th>Dynamic</th>
<th>Share/Combine</th>
<th>Reshare</th>
<th>Storage</th>
<th>Resilience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Herzberg et al. [18]</td>
<td>yes</td>
<td>no</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Cachin et al. [29]</td>
<td>no</td>
<td>no</td>
<td>$O(n^3)$</td>
<td>$O(n^4)$</td>
<td>$O(1)$</td>
<td>$t &lt; n/3$</td>
</tr>
<tr>
<td>Desmedt and Jajodia [30]</td>
<td>yes</td>
<td>yes(^\dagger)</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Wong et al. [31]</td>
<td>yes</td>
<td>yes(^\dagger)</td>
<td>$O(n^2)$</td>
<td>$O(exp(n))$</td>
<td>$O(1)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Baron et al. [32]</td>
<td>yes</td>
<td>yes(^\dagger)</td>
<td>$O(n)$</td>
<td>$O(n^3)$</td>
<td>$O(1)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Opt-CHURP [15]</td>
<td>yes</td>
<td>yes(^\dagger)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Goyal et al. [16]</td>
<td>yes</td>
<td>yes</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$t &lt; n/2$</td>
</tr>
<tr>
<td>Zhou et al. [22]</td>
<td>no</td>
<td>yes(^\dagger)</td>
<td>$O(exp(n))$</td>
<td>$O(exp(n))$</td>
<td>$O(exp(n))$</td>
<td>$t &lt; n/3$</td>
</tr>
<tr>
<td>MPSS [23]</td>
<td>no</td>
<td>yes(^\dagger)</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$t &lt; n/3$</td>
</tr>
<tr>
<td>Exp-CHURP [15]</td>
<td>no</td>
<td>yes(^\dagger)</td>
<td>$O(n^2)$</td>
<td>$O(n^3)$</td>
<td>$O(n)$</td>
<td>$t &lt; n/3$</td>
</tr>
<tr>
<td>COBRA DPSS (this paper)</td>
<td>no</td>
<td>yes</td>
<td>$O(n)$</td>
<td>$O(n^3)$</td>
<td>$O(1)$</td>
<td>$t &lt; n/3$</td>
</tr>
</tbody>
</table>

\(^\dagger\) 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 seen 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 $\Pi$ that can be divided into two subsets: an infinite set of replicas/servers $\Sigma = \{r_1, r_2, \ldots\}$, and an infinite set of clients $\Gamma = \{c_1, c_2, \ldots\}$. 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 $\Pi$ 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 $m_{sk_i}$. A message $m$ encrypted with $pk_i$ is denoted by $E_i(m)$.

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 seen 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].
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) $C = \{\langle D_{1j}, D_{e1}, e_1 \rangle, ..., \langle D_{mj}, D_{em}, e_m \rangle\}$ is the common state, represented by a set of tuples with the information replicated to all servers: $D_{1j}$ 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), $D_{ej}$ corresponds to $D_{1j}$ encrypted using a random symmetric key $K_j$, and $e_j$ is the commitment for the shares of $K_j$;

2) $P_i = \{s_{i1}, ..., s_{im}\}$ is the private state of server $r_i$, containing its shares $s_{ij}$ 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 server.

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., confidential document databases with searchable encryption features [52]), where $D_{1j}$ can be a portion of the encrypted index, $D_{ej}$ is the encrypted data entry, and $s_{1j}$ a share of the encryption key; and (3) outsourced encrypted databases allowing computations directly over the encrypted values [53], where $D_{ej}$ is, for instance, a homomorphically encrypted record and $s_{1j}$ 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 $V$, the adversary can control simultaneously at most $V.t = \lfloor \frac{n-1}{3} \rfloor$ 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.

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 $V_{ini}, ..., V_{cur}$ satisfies Secrecy of its state $S = \{D_1, ..., D_m\}$ if, for each view $V$ and data entry $D_j \in S$ not accessible by faulty clients, an adversary controlling no more than $V.t$ servers of $V$ while $V$ is the current system view, learns no information about $D_j$ besides its non-sensitive part $D_{1j}$.

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 $s_{1j}, ..., s_{mj}$ 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.
recovered its shares to the group; (server locally generates a random polynomial and distributes share recoverable by correct servers, i.e., shares of recovery and reshare. With previous works [15], [23]) is enough for supporting 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 $V$ 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 $(P+R)(k) = P(k) + R(k) = P(k)$.

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 $V_{old}$ to transfer the shared secret to a (possibly overlapping) group $V_{new}$ 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 $V_{old}$ blind their shares with $Q$ and send them to $V_{new}$. 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 \( V_{new} \) obtain renewed shares that, when combined, lead to the same original secret.

Notice that a server of \( V_{new} \) 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 \( V_{new} \).

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):

- \( \{s_i\}_{i\in[n]} \) is a randomized procedure that generates shares \( \{s_i\}_{i\in[n]} \) and a commitment \( P \) of degree \( t \) in which \( P(x) = y \);\(^3\)
- \( \text{false=true} \) is a deterministic procedure that verifies if a share \( s_i \) is in polynomial \( P \) from which \( c \) was generated. If it returns \( \text{true} \), we say \( s_i \) is valid;
- \( y = \text{Reconstruct}(S_{t+1}, x) \) 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 \( V \subset \Sigma \) propose a value \( v \) to all processes in \( V \). All correct processes in \( V \) will eventually output a decision \( v^* \) (Termination), which is the same (Agreement), and that was proposed by some \( r_i \in V \) (Validity). Additionally, we require Validity to be strengthened to ensure the decided value \( v^* \) is useful, i.e., it satisfies some application-specific predicate \( P \).

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

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 \( P(v) = \text{true} \) in every correct process of \( V \). 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 \( P \). 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 \( P \)-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:

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

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

Integrity: If \( A \) cannot corrupt more than \( V, t \) shareholders in the current view \( V \) during Recover, and more than \( V_{old}, t \) servers in \( V_{old} \) and \( V_{new}, t \) servers in \( V_{new} \) 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 \( A \) cannot corrupt more than \( V, t \) shareholders in current view \( V \) during Recover, and more than \( V_{old}, t \) servers in \( V_{old} \) and \( V_{new}, t \) servers in \( V_{new} \) 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: \((S, \{s_i\}_{i \in V}, c) = \text{GeneratePolynomial}(V, x)\) is a randomized protocol that allows shareholders to generate a random polynomial \( P \) of degree \( V.t \) encoding \((x, 0)\). The function outputs at each correct server \( r_i \in V \) 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 \( V.t + 1 \) correct servers obtain valid shares of \( P \), correct servers receiving corrupted shares can verify their shares are corrupted (using \( c \)), and the adversary \( A \) knows no more than \( V.t \) shares of \( P \).

Our implementation of \text{GeneratePolynomial} is described below. The protocol has three phases, as illustrated in Fig. 2:

- **Polynomial Proposal** (Steps S1 and S2),
- **Selection** (Steps S3 and S4),
- **Share Computation** (Step S5).

![PROTOCOL GeneratePolynomial(V, x)]

Server \( r_i \in V \) with input \((V, x)\):

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

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 \( P\text{-Validity} \), 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 \((j, c_j)\) 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 \( O(n) \)-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 \( V \), 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 \( V \), 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.
controls some servers in realization of the approach. For instance, if an adversary
in a non-synchronous setting, which significantly complicates the
communication complexity of the recovery protocol is $O(n^3)$. R1. Requests blinded shares from other servers in V.
R2. Waits for responses.
   a. Upon receiving $V.t + 1$ messages $\langle\text{REC}, s', c, c'\rangle$
      (same c and $c'$) with Verify$(\ast, s', c \odot c') = \text{true},$
      outputs $\langle\text{Reconstruct}(s''_k), k, c\rangle$.
   b. Upon receiving a message $\langle\text{ACC}, j, prop_j, s_{ji}\rangle$, sends it to other servers in V and step R1.

Each server $r_i \in V \setminus \{r_k\}$ with input $(V, k, s_i, c)$:
S1. Upon receiving a blinded share request from $r_k$, runs
      $\text{GeneratePolynomial}(V \setminus \{r_k\}, k)$ to obtain $(S, s'_i, c')$.
S2. If the replica has a share for the secret being recovered, it
      evaluates Verify$(i, s'_i, c')$.
      a) If true, calculates $s''_i = s_i + s'_i$ and sends a message
         $\langle\text{REC}, s''_i, c, c'\rangle$ to $r_k$.
      b) Otherwise, identifies the invalid proposal $prop_j =
         \langle\text{PROPOSAL}, \cdots E_i(s_{ji}) \cdots c_j\rangle$ of a server $r_j$ in
         S and sends $\langle\text{ACC}, j, prop_j, s_{ji}\rangle$ to $r_j$.
S3. Upon receiving a message $\langle\text{ACC}, j, prop_j, s_{ji}\rangle$ from $r_k$, where
      $prop_j = \langle\text{PROPOSAL}, \cdots E_i(s_{ji}) \cdots c_j\rangle$ and
      $r_j \in V \setminus \{r_k\}$, checks if the accusation is sound (i.e., $prop_j$ is
      signed by $r_j$, the encryption of $s_{ji}$ with $r_j$’s public key
      matches $E_i(s_{ji})$, and Verify$(l(s_{ji}), c_j) = \text{false}$). If yes, adds $r_j$
      to an ignore list, otherwise includes $r_j$ in such list.

If $r_i$ receives an invalid share in $\text{GeneratePolynomial}$, it
identifies the invalid proposal(s) that caused this and returns
an accusation to $r_j$. 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.

$\text{Communication complexity}$: 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 resharable 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 $V_{old}$, we might require an execution
of the recovery protocol to remove such servers and create
a renewed share for every correct server of $V_{new}$.

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 $V_{old}$ to blind their stored shares. Servers in $V_{new}$ use $Q'$ shares and the blinded secret $z$ (obtained using
the blinded shares from $V_{old}$) to compute their renewed shares.

In the following, we detail how our protocol reshares a
secret, while “moving” shares from $V_{old}$ to $V_{new}$. The protocol
is divided in three phases: (1) Polynomial Generation (Steps O1,
N1, and N2), (2) Blinded Secret Reconstruction (Steps O2 and
N3) (3) Share Renew (Steps O3 and N4).

Each server $r_i \in V_{old}$ with input $(V_{old}, V_{new}, s_i, c)$:
O1. Executes $\text{GeneratePolynomial}$ to obtain $Q$ and $Q'$
      for $V_{old}$ and $V_{new}$, respectively, and obtains $(S, s''_i, c')$.
O2. If Verify$(i, s''_i, c') = \text{true},$ calculates a blinded share $s''_i =
      s_i + s''_i$ and its commitment $c'' = c \odot c'$, and sends $\langle s''_i, c''\rangle$
      to $V_{new}$.
O3. Deletes $(s_i, c)$ and halts.

Each server $r_j \in V_{new}$ with input $(V_{old}, V_{new})$:
N1. Executes $\text{GeneratePolynomial}$ to obtain $(S, s''_i, c')$.
N2. If Verify$(j, s''_i, c') = \text{false},$ runs $\text{Recover}$ to obtain a valid
      share $s''_i$ of $Q'$.
N3. Waits the reception of $V_{old}.t + 1$ messages $\langle s''_i, c'\rangle$
      with the same $c'$ and Verify$(i, s''_i, c') = \text{true},$ and calculates
      $z = \text{Reconstruct}(\langle s''_i, c'\rangle, 0)$.
N4. Compute the renewed share $s''_i = z - s''_i$ and its corre-
      sponding commitment $c' = c'' \odot c'$. Outputs $\langle s''_i, c'\rangle$.

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

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

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

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

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
PROTOCOL Recover & V, k, \{si\}_i \in V \setminus \{r_k\}, c \\
\hline
Recovering server $r_k \in V$ with input $V$:
R1. Requests blinded shares from other servers in V.
R2. Waits for responses.
   a. Upon receiving $V.t + 1$ messages $\langle\text{REC}, s', c, c'\rangle$
      (same c and $c'$) with Verify$(\ast, s', c \odot c') = \text{true},$
      outputs $\langle\text{Reconstruct}(s''_k), k, c\rangle$. \\
   b. Upon receiving a message $\langle\text{ACC}, j, prop_j, s_{ji}\rangle$, sends it to other servers in V and step R1.
\hline
\end{tabular}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
PROTOCOL Reshare & $V_{old}, V_{new}, \{s_i\}_i \in V_{old}, c$ \\
\hline
Each server $r_i \in V_{old}$ with input $(V_{old}, V_{new}, s_i, c)$:
O1. Executes $\text{GeneratePolynomial}$ to obtain $Q$ and $Q'$
   for $V_{old}$ and $V_{new}$, respectively, and obtains $(S, s''_i, c')$.
O2. If Verify$(i, s''_i, c') = \text{true},$ calculates a blinded share $s''_i =
   s_i + s''_i$ and its commitment $c'' = c \odot c'$, and sends $\langle s''_i, c''\rangle$
   to $V_{new}$.
O3. Deletes $(s_i, c)$ and halts.
\hline
Each server $r_j \in V_{new}$ with input $(V_{old}, V_{new})$:
N1. Executes $\text{GeneratePolynomial}$ to obtain $(S, s''_i, c')$.
N2. If Verify$(j, s''_i, c') = \text{false},$ runs $\text{Recover}$ to obtain a valid
   share $s''_i$ of $Q'$.
N3. Waits the reception of $V_{old}.t + 1$ messages $\langle s''_i, c'\rangle$
   with the same $c'$ and Verify$(i, s''_i, c') = \text{true},$ and calculates
   $z = \text{Reconstruct}(\langle s''_i, c'\rangle, 0)$.
N4. Compute the renewed share $s''_i = z - s''_i$ and its corre-
   sponding commitment $c' = c'' \odot c'$. Outputs $\langle s''_i, c'\rangle$.
\hline
\end{tabular}
\end{table}

4 In this step, $\odot$ represents the inverse operation of $\otimes$, e.g., in Feldman’s
scheme, this is a division of the commitments 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 \(\langle D, D^c, c, \{s_i\}_{i \in \mathcal{V}}\rangle = \text{Update}(\mathcal{V}, D)\) 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 \(\langle D, D^c, c\rangle\) 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’ 3-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 \(D = \text{Read}(\{\mathcal{V}, \mathcal{I}\})\) is implemented by a deterministic protocol that retrieves the confidential data associated with \(\mathcal{I}\) 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 \(\langle C, P_k \rangle = \text{StateRecover}(\mathcal{V}, k, C, \{P_i\}_{i \in \mathcal{V} \setminus \{r_k\}})\) 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 \(\langle C', \{P'_i\}_{i \in \mathcal{V}_{\text{new}}} \rangle = \text{Reconfigure}(\mathcal{V}_{\text{old}}, \mathcal{V}_{\text{new}}, C, \{P_i\}_{i \in \mathcal{V}_{\text{old}}} )\) is a randomized protocol that enables BFT SMR service to change its configuration from \(\mathcal{V}_{\text{old}}\) to \(\mathcal{V}_{\text{new}}\). \(C'\) is the new common state stored by each replica in \(\mathcal{V}_{\text{new}}\), 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 resharding 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 reshard 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 resharin 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 $(k,v)$, 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. 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 $4$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 $O(n^3)$. Furthermore, the fact that CHURP requires $O(n^3)$ 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. Nonetheless, COBRA still achieves a throughput $3 \times$ 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.

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

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) $8 \times$ (resp. $12 \times$) slower. This huge disparity between the two schemes was also observed in previous works [7], [15].

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

*Table II: KV store throughput (operations/sec) for different systems considering 4, 7, and 10 replicas and updates of 1kB.
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.

### D. Reshare

Our third set of experiments considers the latency for resharining 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 resharining 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× 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× 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>BFT-SMaRt</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>VSSR</td>
<td>-</td>
<td>6.9</td>
<td>100.1</td>
<td>5.1</td>
<td>113.1</td>
</tr>
<tr>
<td></td>
<td>COBRA</td>
<td>131.7</td>
<td>2.2</td>
<td>1.1</td>
<td>25.5</td>
<td>160.5</td>
</tr>
<tr>
<td>7</td>
<td>BFT-SMaRt</td>
<td>-</td>
<td>8.4</td>
<td>-</td>
<td>10.3</td>
<td>10.3</td>
</tr>
<tr>
<td></td>
<td>VSSR</td>
<td>-</td>
<td>5.0</td>
<td>18.3</td>
<td>9.9</td>
<td>178.9</td>
</tr>
<tr>
<td></td>
<td>COBRA</td>
<td>141.2</td>
<td>5.0</td>
<td>1.8</td>
<td>21.2</td>
<td>169.2</td>
</tr>
<tr>
<td>10</td>
<td>BFT-SMaRt</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12.3</td>
<td>12.3</td>
</tr>
<tr>
<td></td>
<td>VSSR</td>
<td>-</td>
<td>9.3</td>
<td>231.3</td>
<td>12.9</td>
<td>253.5</td>
</tr>
<tr>
<td></td>
<td>COBRA</td>
<td>136.3</td>
<td>4.6</td>
<td>2.5</td>
<td>25.7</td>
<td>169.1</td>
</tr>
</tbody>
</table>

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 ($t = 32$).
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. resharer) a state with 100 000 secrets in a ten-replicas group 33% (resp. $5 \times$) 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

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 $V_{old}$ obtain shares from $Q$ while servers in $V_{new}$ obtain shares from $Q'$, both with the same random independent term.

### PROTOCOL GeneratePolynomial*($V_{old}, V_{new}$)

Server $r_i \in V_{old}$ with inputs ($V_{old}, V_{new}$):

S1. Invokes $\{(s_{i,1}, \ldots, s_{i,\#V_{old,n}}), c_i\} = \text{Share}(V_{old}, 0, q_i)$ and $\{(\hat{s}_{i,1}, \ldots, \hat{s}_{i,\#V_{new,n}}), c_i\} = \text{Share}(V_{new}, 0, q_i)$, being $q_i$ a random number.

S2. Sends $\langle \text{PROPOSAL}, E_i(s_{i,1}), \ldots, E_{\#V_{old,n}}(s_{i,\#V_{old,n}}), c_i, c_i'\rangle \sigma_i$ to $V_{old}$ and $\langle \text{PROPOSAL}, E_i(\hat{s}_{i,1}), \ldots, E_{\#V_{new,n}}(\hat{s}_{i,\#V_{new,n}}), c_i, c_i'\rangle \sigma_i$ to $V_{new}$.

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

S3. Collects PROPOSAL messages and runs Consensus:

a) Let $r_i \in V_{old}$ be an elected consensus leader. $r_i$ builds a set $S = \{(j, c_j, c_j') : r_i \in V_{old} \text{ using } V_{old,t} + 1 \text{ received valid proposals} \}$ (a proposal from $r_j$ is valid for $r_i$ iff $\langle \text{PROPOSAL}, E_i(s_{j,1}), \ldots, c_j, c_j'\rangle \sigma_j$ is correctly signed and both Verify($l, s_j, c_j$) and SamePol($c_j, c_j'$) returns true) and proposes this set to $V_{old} \cup V_{new}$.

b) A correct replica accepts the set $S$ proposed by $r_i$ if it received valid proposals from all $r_j$ such that $(j, s_j, s_j') \in S$ (the P-Validity predicate).

c) During the protocol execution, a quorum is formed by waiting $V_{old,n} - V_{old,t}$ from $V_{old}$ and $V_{new,n} - V_{new,t}$ from $V_{new}$.

S4. Waits for Consensus to output $S$. For each tuple $(j, c_j, c_j') \in S$, $r_i$ asks for $r_j$’s proposal $\langle \text{PROPOSAL}, \ldots, E_i(s_{j,1}), \ldots, c_j, c_j'\rangle$ in $V_{old} \cup V_{new}$ if it does not have it.

S5. Outputs generated shares:

a) If $r_i \in V_{old}$: decrypts all $E_i(s_{j,1}), \ldots, E_i(s_{j,\#V_{old,n}}), c_j = \bigotimes (j, s_{j,1}, \ldots, s_{j,\#V_{old,n}}, c_j)$, and outputs $\langle \hat{s}_{i,1}, \ldots, \hat{s}_{i,\#V_{old,n}}, \hat{c}_i\rangle$.

b) If $r_i \in V_{new}$: decrypts all $E_i(s_{j,1}), \ldots, E_i(s_{j,\#V_{new,n}}), c_j = \bigotimes (j, s_{j,1}, \ldots, s_{j,\#V_{new,n}}, c_j)$, and outputs $\langle \hat{s}_{i,1}, \ldots, \hat{s}_{i,\#V_{new,n}}, \hat{c}_i\rangle$.

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 $V_{old}$. In S1, instead of generating a polynomial encoding $(x, 0)$ for a given input $x$, each server $r_i \in V_{old}$ 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 $V_{old}$ exchange proposals with encrypted shares of $Q_i$ and send proposals with encrypted shares of $Q'_i$ to $V_{new}$. During consensus (Step S3), servers, besides verifying the validity of their shares, use the predicate SamePol($c_j, c_j'$) 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 $c_j'$. Finally, when running Consensus, instead of waiting for messages from a quorum of $V.n - V.t$ processes before proceeding to the next phase, each server waits for composite quorums containing $V_{old,n} - V_{old,t}$ servers from $V_{old}$ and $V_{new,n} - V_{new,t}$ servers from $V_{new}$. In the end, the protocol ensures that at least $V_{old,t} + 1$ correct servers from $V_{old}$ can reconstruct $Q$ (Step 5a) and $V_{new,t} + 1$ correct servers from $V_{new}$ 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 $F^{DPSS}$, which is built on top of ideal functionalities $F^{VSS}$ and $F^{Pol}$. $F^{DPSS}$ 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, $F^{VSS}$ describes the behavior of a secure VSS scheme with leakage $\mathcal{L}^{VSS}$ and $F^{Pol}$ 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 $\mathcal{L}^{DPSS}, \mathcal{L}^{VSS}$, and $\mathcal{L}^{Pol}$ 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 $F^{DPSS}$, $F^{VSS}$, and $F^{Pol}$, 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 $F^{DPSS}$, Share, Verify, and Reconstruct for $F^{VSS}$, and GeneratePolynomial plus GeneratePolynomial* for $F^{Pol}$. In the real world these actions map to the invocation of protocols $\prod^{DPSS} = \{\text{Share, Verify, Reconstruct, Recover, Reshare}\}$, $\prod^{VSS} = \{\text{Share, Verify, Reconstruct}\}$, and $\prod^{Pol} = \{\text{GeneratePolynomial, GeneratePolynomial*}\}$ 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 $S$, with one simulator for each possible set of correct replicas, that can model all interactions between the
adversary and each functionality, and that these interactions are computationally indistinguishable from its interactions with the respective implementations \( \prod^{V} \) in the real world. Adaptive corruptions are modeled in the CC-adaptive framework by having every simulator \( S_X \in S \) 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 generation polynomial scheme (\( \prod^{VSS} \)). 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 ([8]-[B]), which informally model the guarantees provided by ideal functionalities \( F^{DPSS} \) and \( F^{Pol} \) and their leakage functions \( L^{DPSS} \) and \( L^{Pol} \).

1) Secrecy: As described in [8], COBRA DPSS is composed by the Recover and Reshare protocols, which are assisted by the GeneratePolynomial and GeneratePolynomial* protocols, respectively.

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

- **GeneratePolynomial:**
  - \( \forall r_i \in Corrupt(V) \), shares \( s_{i.1} \) and commitment \( c_s \) of all polynomials \( P \), proposed by correct servers;
  - The set \( S = \{ \langle j, c_j \rangle : r_j \in V \} \) decided by \( V \);
  - \( \forall r_i \in Corrupt(V) \), share \( s_i \) and commitment \( c_i \) of the final polynomial \( P \).

- **GeneratePolynomial*:**
  - \( \forall r_i \in Corrupt(V_{old}) \), shares \( s_{i.1} \) and commitment \( c_s \) of all polynomials \( P \), proposed by correct servers in \( V_{old} \);
  - \( \forall r_i \in Corrupt(V_{new}) \), shares \( s_{i.1} \) and commitment \( c'_s \) of all polynomials \( P' \), proposed by correct servers in \( V_{new} \);
  - The set \( S = \{ \langle j, c_j, c'_j \rangle : r_j \in V_{old} \} \) decided by \( V_{new} \cup V_{old} \);
  - \( \forall r_i \in Corrupt(V_{old}) \), share \( s_i \) and commitment \( c_i \) of the final polynomial \( Q \);
  - \( \forall r_i \in Corrupt(V_{new}) \), share \( s'_i \) and commitment \( c'_i \) of the final polynomial \( Q' \).

**Recover:**

- \( \forall r_i \in Corrupt(V) \), \( \langle S, s_i, c_i \rangle \) resulting from the invocation of GeneratePolynomial;
- \( \forall r_i \in Corrupt(V) \), accusation proposals \( \langle \text{ACC}, j, \text{prop}_j, s_{j.1} \rangle \) sent by \( r_i \);
- If recovering server \( r_k \) is corrupted, blinded polynomial \( B(.) = P(.) + R(.) \), recovered share \( s_{i.1} \), commitments \( c \) and \( c' \), and accusation proposals \( \langle \text{ACC}, j, \text{prop}_j, s_{j.1} \rangle \) sent by servers \( r_i \in V \);

**Reshare:**

- \( \forall r_i \in Corrupt(V_{old}) \), \( \langle S, s_i, c_i \rangle \) resulting from the invocation of GeneratePolynomial*;
- \( \forall r_i \in Corrupt(V_{new}) \), \( \langle S, s_i, c_i \rangle \) resulting from the invocation of GeneratePolynomial*;
- The blinded polynomial \( B(.) = P(.) + R(.) \) and commitment \( c \);
- \( \forall r_j \in V_{new} \) that invokes Recover, and \( \forall r_i \in Corrupt(V_{new}) \), the shares and commitments \( \langle S_j, s'_i, c' \rangle \), the blinded polynomials \( B_j \) if \( r_j \) is also corrupted, and the accusations \( \langle \text{ACC}, k, \text{prop}_k, s_{k.1} \rangle \);
- \( \forall r_i \in Corrupt(V_{new}) \), share \( s_i \) and commitment \( c' \).

To prove secrecy we have the following lemmata.

**LEMMA A.1:** Given a VSS scheme with Hiding and if \( A \) corrupts no more than \( V.t \) servers in \( V \), 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 \( V \) will be randomly generated. Since leader \( r_1 \) defines \( S \) by selecting at least \( V.t + 1 \) of such polynomials sent in valid proposals, there will always be at least one polynomial in \( S \) whose generation cannot be influenced by \( A \). Even if \( r_1 \) is controlled by \( A \), it will still not be able to select \( V.t + 1 \) corrupt proposals for \( S \), since there are at most \( V.t \) corrupted servers in \( V \) 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 \( c_i \) of locally generated polynomials \( P_t \), and hence commitment \( c_i \) of final polynomial \( P \), also reveal no additional information to \( A \). Consequently, \( A \) can not influence the generation of \( P \) and, even though it ends up with \( V.t + 1 \) 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 \( A \) corrupts no more than \( V_{old}.t \) servers in \( V_{old} \) and \( V_{new}.t \) servers in \( V_{new} \), 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 \( A \) can corrupt at most \( V_{old}.t \) servers in \( V_{old} \) and \( V_{new}.t \) servers in \( V_{new} \). 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 \( V_{old} \) invoke Share with a secret and randomly generated value \( q_i \) and by ensuring that \( A \) never learns more than \( V_{old}.t \) shares of polynomial \( Q \) and \( V_{new}.t \) 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 \( c_i \) and \( c'_i \) also reveal no additional information to \( A \) due to the Hiding property of the commitment scheme.

**LEMMA A.3:** If \( A \) corrupts no more than \( V.t \) servers in \( V \) and GeneratePolynomial randomly generates polynomial \( R \), then the information received by \( A \) in Recover is random and independent of secret \( s \).
Lemma A.6: Given a VSS scheme with Binding, if \( A \) corrupts no more than \( V_{old}.t \) servers in \( V_{old} \) and \( V_{new}.t \) servers in \( V_{new} \), 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 \( V_{old}.t+1 \) correct servers in \( V_{old} \) and \( V_{new}.t+1 \) correct servers in \( V_{new} \) 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 \( A \) corrupts no more than \( V.t \) servers in \( V \), and GeneratePolynomial correctly generates a random polynomial \( R \) with \( R(k) = 0 \) that can be reconstructed by at least \( V.t+1 \) correct servers, then, after Recover, recovering server \( r_k \) obtains a share \( s_k = P(k) \) and a new signature \( c' \).

Proof: Consider a server \( r_k \) recovering its share by sending a request to others servers in \( V \). In response, other correct servers execute GeneratePolynomial to generate \( R \) with \( R(k) = 0 \). By Lemma A.6, at least \( V.t+1 \) correct servers will generate shares for \( R \). These servers send blinded shares \( B(i) = P(i) + R(i) \), and \( c' \) to \( r_k \). Server \( r_k \) eventually receives at least \( V.t+1 \) of such messages that are valid and with the same \( c' \). Thus, and assuming the security of the employed commitment scheme, the \( V.t \) corrupted servers in \( V \) 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 verified by \( c' \).

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

Proof: By Lemma A.6, at least \( V_{old}.t+1 \) servers in \( V_{old} \) and \( V_{new}.t+1 \) in \( V_{new} \) obtain valid shares of \( Q \) and...
Lemma A.12: If $A$ corrupts no more than $V_{old}.t + 1$ correct servers in $V_{old}$ and $V_{new}.t$ servers in $V_{new}$, then Reshare terminates.

Proof: By Lemma A.10, GeneratePolynomial+ terminates. By Lemma A.6, at least $V_{old}.t + 1$ correct servers in $V_{old}$ obtain valid shares of polynomial $Q$ and send valid blinded shares to $V_{new}$, terminating the protocol. The remaining servers in $V_{old}$ with invalid shares of $Q$ also terminate the protocol. By Lemma A.6, at least $V_{new}.t + 1$ correct servers in $V_{new}$ obtain valid shares of polynomial $Q'$, and wait for the $V_{old}.t + 1$ valid blinded shares from $V_{old}$. These servers compute their renewed shares and terminate the protocol. The correct servers in $V_{new}$ 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, $A$ 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 $S = \{D_1, \ldots, D_m\}$ is composed of $m$ data entries, with each correct server $r_i$ with a state $S_i = (C_i, P_i)$ (see §III-C). Further, four protocols are supported: Update, Read, StateRecover, and Reconfigure. Besides public data, adversary $A$ has access to the following information:

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

Assuming that $A$ can corrupt at most $V.t + 1$ correct servers in the current view $\mathcal{V}$, and from the security of COBRA DPSS, $A$ 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 $A$ for all stored data entries $D \in S$. 

19