Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols

Soumya Basu[*]
Cornell University; IC3
soumya@cs.cornell.edu

Alin Tomescu[*]
MIT
alinush@mit.edu

Ittai Abraham
VMware Research
iabraham@vmware.com

Dahlia Malkhi[*]
Calibra
dahliamalkhi@gmail.com

Michael K. Reiter
UNC-Chapel Hill; VMware Research
reiter@cs.unc.edu

Emin Gün Sirer
Cornell University; IC3
egs@systems.cs.cornell.edu

Abstract

Byzantine fault tolerant state machine replication (SMR) provides powerful integrity guarantees, but fails to provide any privacy guarantee whatsoever. A natural way to add such privacy guarantees is to secret-share state instead of fully replicating it. Such a combination would enable simple solutions to difficult problems, such as a fair exchange or a distributed certification authority. However, incorporating secret shared state into traditional Byzantine fault tolerant (BFT) SMR protocols presents unique challenges. BFT protocols often use a network model that has some degree of asynchrony, making verifiable secret sharing (VSS) unsuitable. However, full asynchronous VSS (AVSS) is unnecessary as well since the BFT algorithm provides a broadcast channel.

We first present the VSS with share recovery problem, which is the subproblem of AVSS required to incorporate secret shared state into a BFT engine. Then, we provide the first VSS with share recovery solution, KZG-VSSR, in which a failure-free sharing incurs only a constant number of cryptographic operations per replica. Finally, we show how to efficiently integrate any instantiation of VSSR into a BFT replication protocol while incurring only constant overhead. Instantiating VSSR with prior AVSS protocols would require a quadratic communication cost for a single shared value and incur a linear overhead when incorporated into BFT replication.

We demonstrate our end-to-end solution via a a private key-value store built using BFT replication and two instantiations of VSSR, KZG-VSSR and Ped-VSSR, and present its evaluation.

ACM Reference Format:
Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. 2019. Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354207

1 Introduction

Combining the power of Byzantine Fault Tolerant (BFT) replication with secret sharing, one can build a decentralized service that acts over private values in a coordinated manner by consensus decrees. This powerful combination can be leveraged in various ways to build an automated, decentralized threshold trusted third party (T3P) where clients can store private data. For example, it may be used to build a decentralized T3P escrow. Crucially, escrowed secrets will be opened by consensus decree, not necessarily requiring client interaction. This greatly simplifies traditionally difficult problems such as a fair exchange. A fair exchange can be implemented by having one party confidentially store one value, another party confidentially store a second value and a consensus decree that opens both. Another such example is a certification engine, which computes decentralized threshold signatures to certify documents based on some policy. Using polynomial secret sharing, multiple values entrusted to a decentralized T3P may be aggregated without client involvement. Simple additive aggregations are trivial to implement. Arbitrary multi-party computation is possible (e.g., [15, 41]), though more costly.

In all of these use-cases, the enabling core is a mechanism called Verifiable Secret Sharing (VSS) [13] used for populating a decentralized service with secret values. Our use of VSS weaves it into BFT replication in order to automate the handling of secret values. For example, in our private key-value store, a client request to store an entry is broken into two parts, public and private. The public part works as a normal BFT replication request. The private part uses VSS for the client to directly share the private entry. Importantly, replicas delay their participation in the ordering protocol until they obtain a verifiable share of this private entry from the client.

Due to the need to weave VSS into an arbitrary BFT replication engine, the setting of interest to us is asynchronous. A synchronous protocol can simply wait until all honest replicas have received a share, while such a technique breaks in asynchronous settings. It is crucial for the VSS protocol to simply inherit the network model from the BFT replication engine and not require additional assumptions. Thus, we need to solve the problem of *share recovery*, which only prior works in asynchronous VSS (AVSS) schemes have directly addressed.

The best known Asynchronous VSS (AVSS) solutions require a client (dealer) to incur quadratic computation and communication complexities in order to store a single value [1, 29]. This, in turn, imposes a linear computation and storage cost on each replica,

[*] Work done while at VMware Research.

CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354207

which means that the performance overhead increases linearly with the cluster size. When AVSS and BFT replication solutions were originally developed, most BFT solutions were aimed for systems of four (tolerating $f = 1$ faults) or seven ($f = 2$) replicas. However, today, BFT replication is being revisited at scale in blockchain systems of hundreds or thousands of replicas [**? ? ?**]. Incurring such a large degradation in service performance for privacy may be prohibitively expensive.

To incorporate VSS into BFT protocols, this paper introduces a new verifiable secret sharing framework called VSSR. VSSR is a framework that, given a VSS scheme with certain properties, adds share recovery with only a constant factor overhead from the original VSS scheme. We instantiate VSSR in two ways: (1) KZG-VSSR, which uses Kate, et al.'s secret sharing scheme [29] to instantiate a VSS that has constant time share verification and (2) Ped-VSSR, which uses Pedersen's secret sharing scheme [42] which, while only providing linear time share verification and recovery, uses cheaper cryptographic operations and is faster for smaller clusters. Our framework is based on one key concept: the recovery polynomial. The recovery polynomial is a single polynomial that encodes recovery information for f shares. Thus, by only sharing a small, constant number of additional polynomials, the client can enable all $3f + 1$ shares to be recovered.

We intertwine Ped-VSSR and KZG-VSSR in a BFT replication system and build a full private key-value store solution. Our key-value store performs well in practice, with only a 30% to 50% throughput overhead over a non-private key-value store with request latencies less than 35 milliseconds.

This paper contributes a new framework for incorporating VSS into BFT replication systems through the use of recovery polynomials, VSSR. We then instantiate VSSR using two verifiable secret sharing schemes and benchmark the overhead that our new framework adds. Finally, we incorporate our two instantiations into PBFT [12] and evaluate a private, Byzantine Fault Tolerant key value store.

2 Technical Overview

In this section, we provide a high-level, informal overview of the core technique we use to solve the VSS with share recovery problem. We first introduce the asynchronous VSS (AVSS) problem and show how share recovery arises in the AVSS setting and its relation to Byzantine Fault Tolerance. We then discuss some prior works in AVSS and common techniques for solving the share recovery problem. Finally, we provide a high level overview of our contribution.

2.1 The Asynchronous VSS Problem

In the asynchronous VSS problem, a dealer shares to a group of n participants a polynomial s. The API for sharing is denoted by vssShare. If the sharing completes anywhere, then eventually every non-faulty participant completes the sharing. The basic method for secret sharing is to provide participant i a point $(x_i, s(x_i))$ on the polynomial s. The method fulfills two key properties, *hiding* and *binding*.

Loosely speaking, hiding means that for a polynomial s of degree f, any $k = f + 1$ shares suffice to reconstruct it via interpolation (API: vssReconstruct), and that no combination of f or less reveal any information about it. Binding means that every participant receives, in addition to its private share, a global commitment c to the polynomial s that binds the share it receives as a verifiable valid share of s (API: vssVerify).

In asynchronous settings, a dealer can wait for at most $n - f$ participants to acknowledge receiving a valid share, before it inevitably may walk away. Note that it is possible for the dealer to walk away before all of the honest replicas have a valid share. The asynchronous VSS problem requires that if the dealer (or any participant) completes the share protocol, then every correct participant can eventually reconstruct its share using a distributed protocol with $f + 1$ correct participants: Participants contribute recovery information (API: vssRecoverContrib*), which is validated by the recipient (API: vssRecoverVerify*) and then combined to reconstruct the missing share (API: vssRecover*).

VSS in Byzantine Fault Tolerance Incorporating VSS into Byzantine Fault Tolerance (BFT) presents some unique challenges. BFT protocols have a large variety of network assumptions and consequently have to handle the scenario where a replica is missing part of the state. In traditional BFT protocols, the replicated state is identical across replicas, making state recovery a matter of transmitting and validating the state. With secret shared state, each replica stores a unique share of the replicated private state, which makes the problem of share recovery much more complicated. AVSS protocols have minimal assumptions on the network and must solve an analogous version of share recovery.

There are a few key design goals to meet when incorporating AVSS into BFT. For example, it is acceptable for a Byzantine client to lose the hiding guarantee. However, every sharing must always be binding, since otherwise the replicated state machine can be in an inconsistent state. Typical AVSS schemes need a reliable broadcast protocol to guarantee this binding property. However, state machine replication (SMR) also solves the reliable broadcast problem, eliminating the need for VSSR to implement reliable broadcast. In particular, VSSR does not make any assumptions about the network itself.

Additionally, there are many different Byzantine Fault Tolerance (BFT) algorithms in the literature that have been optimized to perform under certain circumstances. For example, some BFT algorithms [32**?**] have often incorporated a linear "fast-path" suitable for cases where there are few failures. In particular, this search for more optimized performance in the common case is something that we foresee continuing in the BFT literature.

Thus, it is important for a secret sharing scheme to have minimal overhead in the common case. In particular, a verifiable secret sharing scheme used in BFT must meet the requirement that sharing a secret only requires replicas to compute a constant number of cryptographic operations. This ensures that the same techniques will be reusable for more scalable BFT protocols that work with larger clusters.

2.2 Existing Solutions

The seminal work by Shamir [45] introduced the idea of employing polynomial interpolation, a technique that was used before for error correction codes, to share a secret with unconditional security. A line of work emanated from this result and addressed many additional features, such as share verifiability, asynchrony, and proactive share refresh.

Share verifiability tackles the problem of a malicious dealer that equivocates and maliciously shares values that are inconsistent. There are many such schemes with different properties, from classical works such as Feldman's [20] and Pedersen's [42] schemes to newer works such as Kate et al. [29] and SCAPE [11]. VSSR can take many of these works as input and construct a verifiable secret sharing scheme that provides share recovery.

Original solutions for asynchronous VSS in the information-theoretic setting were introduced in the context of Byzantine agreement and secure MPC [10]. They incur communication complexity of $O(n^6 \log n)$ and message complexity $O(n^5)$.

AVSS. The first practical asynchronous VSS solution in the computational setting was introduced by Cachin et al. [8]. We will refer to it by the name AVSS. To handle share recovery, AVSS uses a bivariate secret polynomial $\hat{s}(\cdot,\cdot)$. Share i consists of two univariate polynomials, $\hat{s}(i,\cdot)$, $\hat{s}(\cdot,i)$, and so the dealer sends $O(n)$ information to each participant. A missing i'th share can be reconstructed from $f+1$ evaluations of $\hat{s}(i,\cdot)$ and $f+1$ evaluations of $\hat{s}(\cdot,i)$, incurring linear communication overhead per recovery, for an overall recovery complexity of $O(n^2)$ messages and $O(n^3)$ bits.

Additionally, participants need to verify that all shares are bound to the same polynomial. AVSS makes use of Pedersen polynomial commitments [42] for all polynomials $\hat{s}(i,\cdot)$, $\hat{s}(\cdot,i)$, $i = 1..n$. This commitment scheme leverages the hardness of discrete log in a multiplicative group of order q with generator g. A commitment $c(v)$ to a value $v \in \mathbb{Z}_q$ is a value $g^v h^r$, where h is another element of the group and r is a secret drawn at random from $\mathbb{Z}_q$. A Pedersen commitment to a polynomial $s(\cdot) \in \mathbb{Z}_q[x]$ consists of a set of commitments to n values, i.e., $c(s(\cdot)) = \{\langle x_i, c(s(x_i))\rangle\}_{i=1}^n$. Given any pair $\langle x, s(x)\rangle$, it is possible to verify that this point is on $s(\cdot)$ using the commitment's homomorphic properties, i.e., that for any $v_1, v_2 \in \mathbb{Z}_q$, $c(v_1)c(v_2)$ is a valid commitment to $v_1 + v_2 \bmod q$.

AVSS weaves into the sharing protocol the dissemination of commitments [7] while incurring message complexity $O(n^2)$ and communication complexity $O(n^3)$.

eAVSS-SC. Kate et al. [29] introduces a polynomial commitment that has constant size. This commitment scheme leverages the hardness of the q-Strong Diffe-Hellman assumption in some group with order p where g is a generator. In Kate et al., a commitment $c(s(\cdot))$ is defined as $c(s(\cdot)) = g^{s(\tau)}$, where τ is unknown to all participants and $s(\tau)$ is the polynomial s evaluated at τ. To commit to a particular evaluation (or share) $s(i)$, the dealer also produces a witness, $g^{\frac{s(\tau)-s(i)}{\tau-i}}$. Given any triple of share, witness and commitment, it is possible to verify that the share is indeed the evaluation of the polynomial at that point using a bilinear map. The technique was employed by Backes et al. [3] to construct an asynchronous VSS scheme called eAVSS-SC that incurs both message and communication complexities $O(n^2)$.

In eAVSS-SC, a dealer chooses, in addition to the secret polynomial s, another n polynomials $\hat{s}_i$, $i = 1..n$. $\hat{s}_i$ encodes share i of s for recovery purposes. Each of s, $\hat{s}_i$, has a constant-size polynomial commitment due to the scheme by Kate et al. [29]. The commitments are constructed such that a commitment of $\hat{s}_i$ validates it as a share of s. Using the homomorphism of the commitments, eAVSS-SC weaves into the sharing protocol the dissemination of commitments while incurring both message complexity and bit complexity $O(n^2)$.

2.3 Share Recovery Techniques

AVSS schemes, like the above, and proactive secret sharing [28] present possible solutions to the problem of verifiable secret sharing with share recovery. However, these schemes are not amenable for use in Byzantine Fault Tolerance protocols.

In prior AVSS schemes, the dealer computes $O(n^2)$ commitment values and sends $O(n^2)$ bits. Hence, to date, all asynchronous VSS solutions require a dealer storing a single secret to incur $O(n^2)$ communication complexities, which means that a replica's CPU must typically perform $O(n)$ cryptographic operations for a single sharing. This can be quite prohibitive for moderate n values, e.g., $n = 1,000$, and infeasible for $n = 50,000$. Fundamentally, prior asynchronous VSS schemes allow share recovery by having the dealer enumerate all pairwise responses between replicas during recovery. In other words, if replica i is helping replica $\hat{i}$ recover, the dealer has shared with i the response to send to $\hat{i}$. This must necessarily incur quadratic bandwidth on the dealer and linear CPU overhead on each replica.

Prior proactive secret sharing schemes (PSS) offer other approaches to share recovery. Some PSS schemes have the dealer enumerate all pairwise responses through the use of bivariate polynomials [34, 40]. This approach shares the same performance overhead and weaknesses as state-of-the-art AVSS schemes. Other PSS schemes use a resharing technique [6, 39], where each replica first constructs a secret sharing of their original share and then reconstructs a new share through interpolation. While this has low overhead in the sharing phase, this forces every replica to obtain a new share whenever a single replica needs to initiate share recovery. A third technique employed by PSS schemes is batching [4, 5], where the cost is still quadratic but $O(n)$ shares can be shared in parallel. Batching was also leveraged by a recent AVSS protocol to have a linear amortized overhead [1]. While batching has the same asymptotic overheads as KZG-VSSR in a failure free environment, it requires waiting for $O(n)$ shares which incurs high latency when n is large.

VSSR uses a different approach, where we make use of a distributed pseudorandom function (DPRF) and generate recovery responses dynamically using information shared in the setup phase. Table 1 compares the bandwidth costs of the different ways to instantiate VSSR.

2.4 VSSR

Our solution, named VSSR (for 'VSS with Recovery'), is the first in which the replica work is constant per sharing in the failure free case. In the worst case, when there are f participant failures or the dealer is Byzantine, the overhead is still quadratic as in prior schemes. We present a strawman solution that introduces the key insight behind VSSR and describe how our final solution fixes the issues present in the strawman.

Our strawman uses proactive secret sharing [28] to help a participant re-compute the original share that they were given. Informally, suppose that a replica i has share $s(i)$, which is simply a point on the

Technique	Sharing Phase		Recovery Phase
	Client	Replica	Replica
Bivariate Polynomials [40]	$O(n^2)$	$O(n)$	$O(n^2)$
Resharing [34]	$O(n)$	$O(1)$	$O(n^2)$
Batching [1]	$O(n^2)$	$O(n)$	$O(n^2)$
KZG-VSSR	$O(n)$	$O(1)$	$O(n)$
Ped-VSSR	$O(n^2)$	$O(n)$	$O(n^2)$

Table 1: Communication overhead of possible VSSR designs in the typical scenario when the dealer is honest. n denotes the number of replicas. For the recovery phase, we present the cost of the recovering replica.

polynomial s. In order to recover this share, we are given a *Recovery Polynomial* (RP), which is random at every point except for i, where $s_i(i) = 0$. Thus, if the recovering replica i receives shares of the sum of the original polynomial and the masking polynomial, $s_i(\cdot) + s(\cdot)$, it can recover its own share without obtaining any information about any other share.

The strawman solution assumes that the recovery polynomial is given for free. We solve this by having the dealer construct the recovery polynomials and sharing it with the replicas. The system assumes optimistically that the dealer is honest in sharing the recovery polynomials. If, later on, the dealer is found to be dishonest, then the privacy guarantee for the dealer is broken and the secret is revealed. At this point, our scheme still pays a quadratic overhead in the common case.

The second problem our strawman solution has is that it requires one recovery polynomial for each share. This means that the dealer must construct n recovery polynomials and incur a quadratic overhead. We bring this overhead down by a factor of f, which equals $\frac{n}{3}$ in our case, by allowing f shares to be recovered by a single recovery polynomial. We do this by removing the constraint that $s_i(i) = 0$ and using a distributed pseudorandom function in order to communicate the value of $s_i(i)$ efficiently. This means that the dealer constructs four recovery polynomials, which is a constant factor overhead to the original VSS scheme.

We now present a high level description of our final protocol. To share a secret, the dealer partitions the secret shares of s into $\ell = \lceil n/f \rceil$ groups, and uses ℓ recovery polynomials (RPs) s_j, $j = 1..\ell$, to encode the corresponding groups. Every one of the original n shares of s is encoded in one of the RPs. The dealer shares both s and the s_j's among the n participants, and participants use the s_j's for share recovery.

More specifically, an RP s_j is a random polynomial of degree f that has f pre-defined points. For $(j-1)f \leq i < jf$, the recovery polynomial s_j is constructed so that $s_j(i) = y_i$, where $y_i = \mathcal{F}(i)$ for a DPRF $\mathcal{F}$ with reconstruction threshold f. In our actual construction, $y_i = \mathcal{F}(\langle r, i \rangle)$ for a random value r, to ensure that s_j is distinct for each sharing, but we elide r for our discussion here.

To recover its share, participant i probes other participants, to which each participant responds with its share of $s + s_j$ for $j = \lceil i/f \rceil$. Each participant $\hat{i}$ can construct its response from its shares of s and s_j. In addition, participant $\hat{i}$ also responds with their shares of $\mathcal{F}(i)$, i.e., of the secret value $s_j(i)$. (In API terms, vssRecoverContrib* returns a share of $s + s_j$ and a share of $\mathcal{F}(\hat{i})$.) Participant $\hat{i}$ then reconstructs $s + s_j$ in full and computes $(s + s_j)(i) - \mathcal{F}(i) = s(i)$ (API: vssRecover*).

To verify a recovery share (vssRecoverVerify*), participant i first validates each share of $s + s_j$ that it receives against the commitment $c(s + s_j)$, which it computes from $c(s)$ and $c(s_j)$. Then it validates the recovery result against the commitment $c(s)$.

If validation fails, then participant i can prove to the other participants that the dealer is bad. In that case, different from AVSS and eAVSS-SC, participants expose the dealer's secret.

The complexities incurred by different participants at different steps of the VSSR protocol instantiated with Kate et al. [29] are as follows. A dealer provides each of n participants shares and constant-size commitments on $\ell + 1$ polynomials. The total communication complexity is $O(\ell n)$, or simply $O(n)$ since in our setting, ℓ is four. When a participant receives a recovery request, it sends a constant amount of information to the requester, for a total $O(t)$ communication for t requests. Finally, each participant requiring share recovery obtains shares from other participants incurring $O(n)$ communication, for a total $O(tn)$ communication for t requests.

3 Share Recovery in Verifiable Secret Sharing

In this section, we detail our VSS protocol and its security. We begin with the definitions of distributed pseudorandom functions (Section 3.1) and verifiable secret sharing (Section 3.2). We will then detail our goals (Section 3.3), further assumptions on which our scheme builds (Section 3.4), our construction (Section 3.5) and its security (Section 3.6). Finally, we show how to instantiate our implementation (Section 3.7).

Note that our proofs are applicable to *any* schemes that satisfy our descriptions below. To highlight the generality of our descriptions, we instantiate our secret sharing scheme described in Section 3.2 in two ways [29, 42], of which one gives us the desired asymptotic complexity while the other uses more inexpensive cryptographic operations.

3.1 Distributed Pseudorandom Functions

A distributed pseudorandom function (DPRF) is a pseudorandom function that requires the cooperation of k replicas out of n total replicas to evaluate [38]. A DPRF $\mathcal{F}$ provides the following interfaces, where $[n] = \{1, \ldots, n\}$.

- dprfInit is a randomized procedure that returns a set of pairs $\{\langle dpk_i, dsk_i \rangle\}_{i \in [n]} \leftarrow$ dprfInit$(1^\kappa, k, n, \mathbf{D}, \mathbf{R})$. Each dpk_i is public key, and each dsk_i is its corresponding private key.
- dprfContrib is a randomized procedure that returns a *contribution* $d \leftarrow$ dprfContrib(dsk_i, x) if $x \in \mathbf{D}$ and failure $(\perp)$ otherwise.
- dprfVerify is a deterministic procedure that returns a boolean value. We require that dprfVerify(dpk_i, x, d) returns true if d is output from dprfContrib(dsk_i, x) with nonzero probability, for the private key dsk_i corresponding to dpk_i.
- dprfEval is a deterministic procedure that returns a value $y \leftarrow$ dprfEval$(x, \{d_i\}_{i \in I})$, where $y \in \mathbf{R}$, if $x \in \mathbf{D}$, $|I| \geq k$ and for all $i \in I$, dprfVerify(dpk_i, x, d_i) returns true. Otherwise,

$\mathsf{dprfEval}(x, \{d_i\}_{i \in I})$ returns $\perp$.

Security for a distributed pseudorandom function (DPRF), informally, is defined by an adversary that is unable to distinguish the output from the DPRF from a random oracle even if it has the ability to compromise any set of $k - 1$ replicas.

More formally, security for a distributed pseudorandom function is defined as follows. An adversary $\mathcal{A}_{\mathcal{F}}$ is provided inputs $\langle dpk_i \rangle_{i \in [n]}$, k, n, $\mathbf{D}$, and $\mathbf{R}$, where $\{\langle dpk_i, dsk_i \rangle\}_{i \in [n]} \leftarrow \mathsf{dprfInit}(1^\kappa, k, n, \mathbf{D}, \mathbf{R})$. In addition, $\mathcal{A}_{\mathcal{F}}$ is given oracle access to $n+1$ oracles. The first n oracles, denoted $\langle O_{\mathcal{F},i} \rangle_{i \in [n]}$, each supports two types of queries. $\mathcal{A}_{\mathcal{F}}$ can invoke $O_{\mathcal{F},i}.\mathit{contrib}(x)$, which returns $\mathsf{dprfContrib}(dsk_i, x)$, or it can invoke $O_{\mathcal{F},i}.\mathit{compromise}$, which returns dsk_i. The last oracle provided to $\mathcal{A}_{\mathcal{F}}$ is denoted $O^{?}_{\mathcal{F}} : \mathbf{D} \to \mathbf{R}$ and is instantiated as one of two oracles, either $O^{\text{real}}_{\mathcal{F}}$ or $O^{\text{rand}}_{\mathcal{F}}$. Oracle $O^{\text{real}}_{\mathcal{F}}$, on input x, selects a subset $I \subseteq [n]$ at random of size $|I| = k$, invokes $d_i \leftarrow O_{\mathcal{F},i}.\mathit{contrib}(x)$ for each $i \in I$, and returns $\mathsf{dprfEval}(x, \{d_i\}_{i \in I})$. Oracle $O^{\text{rand}}_{\mathcal{F}}$ is instantiated as a function chosen uniformly at random from the set of all functions from $\mathbf{D}$ to $\mathbf{R}$. For any $x \in \mathbf{D}$, let I_x be the oracle indices such that for each $i \in I_x$, $\mathcal{A}_{\mathcal{F}}$ invokes $O_{\mathcal{F},i}.\mathit{compromise}$ or $O_{\mathcal{F},i}.\mathit{contrib}(x)$. Then, $\mathcal{A}_{\mathcal{F}}$ is *legitimate* if $|I_x| < k$ for every x for which $\mathcal{A}_{\mathcal{F}}$ invokes $O^{?}_{\mathcal{F}}(x)$. Finally, $\mathcal{A}_{\mathcal{F}}$ outputs a bit. We say that the distributed pseudorandom function is secure if for all legitimate adversaries $\mathcal{A}_{\mathcal{F}}$ that run in time polynomial in κ,

$$\begin{aligned} &\mathbb{P}\left(\mathcal{A}_{\mathcal{F}}^{\langle O_{\mathcal{F},i} \rangle_{i \in [n]}, O^{\text{real}}_{\mathcal{F}}}(\langle dpk_i \rangle_{i \in [n]}, k, n, \mathbf{D}, \mathbf{R}) = 1\right) \\ - &\mathbb{P}\left(\mathcal{A}_{\mathcal{F}}^{\langle O_{\mathcal{F},i} \rangle_{i \in [n]}, O^{\text{rand}}_{\mathcal{F}}}(\langle dpk_i \rangle_{i \in [n]}, k, n, \mathbf{D}, \mathbf{R}) = 1\right) \end{aligned} \tag{1}$$

is negligible in κ.

3.2 Verifiable Secret Sharing

Verifiable Secret Sharing (VSS) is a way to share a secret so that it requires a coalition of k replicas out of n total replicas in order to reconstruct the secret. A VSS scheme provides the following interfaces:

- $\mathsf{vssInit}$ is a randomized procedure that returns $\langle q, \{\langle vpk_i, vsk_i \rangle\}_{i \in [n]} \rangle \leftarrow \mathsf{vssInit}(1^\kappa, k, n)$. Here, q is a prime of length κ bits. Each vpk_i is a public key, and each vsk_i is its corresponding private key.
- $\mathsf{vssShare}$ is a randomized procedure that produces $\langle c, \{u_i\}_{i \in [n]} \rangle \leftarrow \mathsf{vssShare}(s, q, \{vpk_i\}_{i \in [n]})$. Here, $s \in \mathbb{Z}_q[x]$ is a degree $k - 1$ polynomial, and q and $\{vpk_i\}_{i \in [n]}$ are as output by $\mathsf{vssInit}$. The value c is a *commitment*, and each u_i is a *share*.
- $\mathsf{vssVerify}$ is a deterministic procedure that returns a boolean. We require that $\mathsf{vssVerify}(vpk_i, c, u_i)$ return true if $\langle c, u_i \rangle$ (i.e., with arbitrary $\{u_{\hat{i}}\}_{\hat{i} \neq i}$) is output from $\mathsf{vssShare}(s, q, \{vpk_i\}_{i \in [n]})$ with nonzero probability.
- $\mathsf{vssReconstruct}$ is a deterministic procedure that returns a value $s \leftarrow \mathsf{vssReconstruct}(c, \{\langle vpk_i, u_i \rangle\}_{i \in I})$ where $s \in \mathbb{Z}_q[x]$ of degree $k - 1$, if $|I| \geq k$ and for all $i \in I$, $\mathsf{vssVerify}(vpk_i, c, u_i)$ returns true. Otherwise, $\mathsf{vssReconstruct}(c, \{\langle vpk_i, u_i \rangle\}_{i \in I})$ returns $\perp$.

The security of a VSS scheme lies in its *hiding* and *binding* properties.

3.2.1 Hiding Informally, the hiding property is set up in the following way. Suppose you have an adversary that gave the client two secrets, of which the client picked one randomly and shared it using our VSS scheme. Even with access to $k - 1$ of the replicas and their secrets, the adversary cannot tell which secret was shared with high probability. Note that our hiding definition is computational since we would like our transformation to be as ambivalent to the underlying secret sharing scheme as possible. Any information theoretic secret sharing scheme would satisfy these requirements as well.

More formally, a hiding adversary $\mathcal{A}_{\mathcal{V}}$ is provided inputs q and $\{vpk_i\}_{i \in [n]}$, where $\langle q, \{\langle vpk_i, vsk_i \rangle\}_{i \in [n]} \rangle \leftarrow \mathsf{vssInit}(1^\kappa, k, n)$, and access to $n + 1$ oracles. The first n oracles are denoted $\langle O_{\mathcal{V},i} \rangle_{i \in [n]}$; each $O_{\mathcal{V},i}$ is initialized with vsk_i and can be invoked as described below. The last oracle provided to $\mathcal{A}_{\mathcal{V}}$ is denoted $O^{b}_{\mathcal{V}}$, where $b \in \{0, 1\}$. $\mathcal{A}_{\mathcal{V}}$ can invoke this oracle with two inputs $s_0, s_1 \in \mathbb{Z}_q$. When invoked, $O^{b}_{\mathcal{V}}$ generates a random $\hat{s} \in \mathbb{Z}_q[x]$ of degree $k-1$ such that $\hat{s}(0) = s_b$ and performs $\langle c, \{u_i\}_{i \in [n]} \rangle \leftarrow \mathsf{vssShare}(\hat{s}, q, \{vpk_i\}_{i \in [n]})$, providing c to $\mathcal{A}_{\mathcal{V}}$ and $\langle c, u_i \rangle$ to $O_{\mathcal{V},i}$. The oracles $\langle O_{\mathcal{V},i} \rangle_{i \in [n]}$ can be invoked by $\mathcal{A}_{\mathcal{V}}$ as follows. $\mathcal{A}_{\mathcal{V}}$ can invoke $O_{\mathcal{V},i}.\mathit{contrib}(c)$, which returns the share u_i provided to $O_{\mathcal{V},i}$ with commitment c by $O^{b}_{\mathcal{V}}$. $\mathcal{A}_{\mathcal{V}}$ can also invoke $O_{\mathcal{V},i}.\mathit{compromise}$, which returns vsk_i and all $\langle c, u_i \rangle$ pairs received from $O^{b}_{\mathcal{V}}$. For any c, let I_c be the oracle indices such that for each $i \in I_c$, $\mathcal{A}_{\mathcal{V}}$ invokes $O_{\mathcal{V},i}.\mathit{compromise}$ or $O_{\mathcal{V},i}.\mathit{contrib}(c)$. Then, $\mathcal{A}_{\mathcal{V}}$ is *legitimate* if $|I_c| < k$ for every c. Finally, $\mathcal{A}_{\mathcal{V}}$ outputs a bit. We say that the VSS $\mathcal{V}$ is *hiding* if for all legitimate adversaries $\mathcal{A}_{\mathcal{V}}$ that run in time polynomial in κ,

$$\begin{aligned} &\mathbb{P}\left(\mathcal{A}_{\mathcal{V}}^{\langle O_{\mathcal{V},i} \rangle_{i \in [n]}, O^{1}_{\mathcal{V}}}(q, \{vpk_i\}_{i \in [n]}) = 1\right) \\ - \quad &\mathbb{P}\left(\mathcal{A}_{\mathcal{V}}^{\langle O_{\mathcal{V},i} \rangle_{i \in [n]}, O^{0}_{\mathcal{V}}}(q, \{vpk_i\}_{i \in [n]}) = 1\right) \end{aligned} \tag{2}$$

is negligible in κ.

3.2.2 Binding Informally, the binding property says that an adversarial dealer's public commitment uniquely identifies the secret that is reconstructed. Essentially, the probability that the dealer can cause two different secrets to be reconstructed with the same public commitment is negligible.

Formally, a binding adversary $\mathcal{A}_{\mathcal{V}}$ is provided inputs $\langle q, \{\langle vpk_i, vsk_i \rangle\}_{i \in [n]} \rangle \leftarrow \mathsf{vssInit}(1^\kappa, k, n)$. $\mathcal{A}_{\mathcal{V}}$ outputs c, $\{u_i\}_{i \in I}$ and $\{\hat{u}_i\}_{i \in \hat{I}}$. We say that VSS $\mathcal{V}$ is *binding* if for all binding adversaries $\mathcal{A}_{\mathcal{V}}$ that run in time polynomial in κ,

$$\mathbb{P}\left(\begin{array}{ll} & \mathsf{vssReconstruct}(c, \{\langle vpk_i, u_i \rangle\}_{i \in I}) = s \\ \wedge & \mathsf{vssReconstruct}(c, \{\langle vpk_i, \hat{u}_i \rangle\}_{i \in \hat{I}}) = \hat{s} \\ \wedge & s \neq \perp \wedge \hat{s} \neq \perp \wedge s \neq \hat{s} \end{array}\right)$$

is negligible in κ, where the probability is taken with respect to random choices made in $\mathsf{vssInit}$ and by $\mathcal{A}_{\mathcal{V}}$.

3.3 Goals

Given such a VSS scheme $\mathcal{V}$ and a DPRF $\mathcal{F}$, our goal is to construct a new VSS scheme $\mathcal{V}^*$ that provides the $\mathsf{vssInit}$, $\mathsf{vssShare}$, $\mathsf{vssVerify}$, and $\mathsf{vssReconstruct}$ algorithms (denoted $\mathsf{vssInit}^*$, $\mathsf{vssShare}^*$, $\mathsf{vssVerify}^*$ and $\mathsf{vssReconstruct}^*$ for $\mathcal{V}^*$, respectively) as defined in Section 3.1, as well as three more algorithms, denoted $\mathsf{vssRecoverContrib}^*$, vssRecoverVerify*, and vssRecover*. We allow the vssShare* algorithm to accept additional arguments (a set of private keys for a DPRF) and to return an additional value r that is provided as input to all procedures except for vssInit*. The algorithms vssRecoverContrib*, vssRecoverVerify*, and vssRecover* together permit a replica to recover its share from other replicas, and behave as follows:

- vssRecoverContrib* is a randomized procedure that returns $v_i^* \leftarrow \mathsf{vssRecoverContrib}^*(c^*, r, vsk_i^*, u_i^*, \hat{\imath})$ where v_i^* is a *recovery share* with properties described below.
- vssRecoverVerify* is a deterministic procedure that returns a boolean. $\mathsf{vssRecoverVerify}^*(c^*, r, v_i^*, vpk_i^*, \hat{\imath})$ must return true if v_i^* is output from $\mathsf{vssRecoverContrib}^*(c^*, r, vsk_i^*, u_i^*, \hat{\imath})$ with nonzero probability and $\mathsf{vssVerify}^*(vpk_i^*, c^*, r, u_i^*)$ returns true.
- vssRecover* is a deterministic procedure that returns $u_{\hat{\imath}}^* \leftarrow \mathsf{vssRecover}^*(c^*, r, \{\langle vpk_i^*, v_i^*\rangle\}_{i\in I}, \hat{\imath}, vpk_{\hat{\imath}}^*)$ if $|I| \geq k$, $\mathsf{vssRecoverVerify}^*(c^*, r, v_i^*, vpk_i^*, \hat{\imath})$ returns true for all $i \in I$, and $\mathsf{vssVerify}^*(vpk_{\hat{\imath}}^*, c^*, r, u_{\hat{\imath}}^*)$ returns true. Otherwise, $\mathsf{vssRecover}^*(c^*, r, \{\langle vpk_i^*, v_i^*\rangle\}_{i\in I}, \hat{\imath}, vpk_{\hat{\imath}}^*)$ returns $\perp$.

Due to the additional interfaces above, we change the definition of hiding security as follows. Each oracle $O_{\mathcal{V}^*,i}$ additionally supports a query $O_{\mathcal{V}^*,i}.recover(c^*, \hat{\imath})$ that returns $v_i^* \leftarrow \mathsf{vssRecoverContrib}^*(c^*, r, vsk_i^*, u_i^*, \hat{\imath})$. For any c^*, let I_{c^*} be the oracle indices such that for each $i \in I_{c^*}$, $\mathcal{A}_{\mathcal{V}^*}$ invokes $O_{\mathcal{V}^*,i}.compromise$, $O_{\mathcal{V}^*,i}.contrib(c^*)$, or $\{O_{\mathcal{V}^*,\hat{\imath}}.recover(c^*, i)\}_{\hat{\imath}\in\hat{I}}$ where $|\hat{I}| \geq k$. Then, $\mathcal{A}_{\mathcal{V}^*}$ is *legitimate* if $|I_{c^*}| < k$ for every c^*.

3.4 Assumptions on Underlying VSS

Our construction combines an existing VSS scheme with a DPRF for which, if $\langle q, \{\langle vpk_i, vsk_i\rangle\}_{i\in[n]}\rangle \leftarrow \mathsf{vssInit}(1^\kappa, k, n)$, then $\mathbf{R} = \mathbb{Z}_q$ and each share u_i output from vssShare is in $\mathbb{Z}_q$. In addition, we require that the VSS offer additional procedures, as follows.

- There is a procedure vssMakeSecret that creates
$$s \leftarrow \mathsf{vssMakeSecret}(q, \{\langle x_i, y_i\rangle\}_{i\in I})$$
where $s \in \mathbb{Z}_q[x]$ is of degree $|I|$, and so that if
$$\langle c, \{u_i\}_{i\in[n]}\rangle \leftarrow \mathsf{vssShare}(s, q, \{vpk_i\}_{i\in[n]})$$
then $u_i = y_i$ for any $i \in I$.
- There is a procedure vssCombineCommitments such that if
$$\mathsf{vssReconstruct}(c, \{\langle vpk_i, u_i\rangle\}_{i\in I}) = s$$
$$\mathsf{vssReconstruct}(\hat{c}, \{\langle vpk_i, \hat{u}_i\rangle\}_{i\in I}) = \hat{s}$$
where $s, \hat{s} \neq \perp$, and if
$$\check{c} \leftarrow \mathsf{vssCombineCommitments}(c, \hat{c})$$
then
$$\mathsf{vssReconstruct}(\check{c}, \{\langle vpk_i, (u_i + \hat{u}_i)\rangle\}_{i\in I}) = s + \hat{s}$$

An example of such a scheme is that due to Pedersen [42].

3.5 VSS Scheme with Recovery

Below we describe the procedures that make up the VSS scheme $\mathcal{V}^*$. The algorithms are expressed in terms of constants n (the number of replicas), k (the reconstruction threshold), and $\ell = \lceil n/(k-1)\rceil$. Each share u_i^* and commitment c^* is a zero-indexed vector of $\ell + 1$ elements. We denote the j-th element of each by $u_i^*[j]$ and $c^*[j]$, respectively, for $0 \leq j \leq \ell$. Line numbers below refer to Figure 1.

vssInit* initializes the underlying VSS $\mathcal{V}$ in line 2, as well as a DPRF $\mathcal{F}$ in line 3. The public key vpk_i^* for replica i consists of its public key vpk_i for $\mathcal{V}$ and its public key dpk_i for $\mathcal{F}$ (line 7) and similarly for the private key vsk_i^* (line 6).

vssShare* is modified to take in all of the private keys $\{dsk_i\}_{i\in[n]}$ for the DPRF $\mathcal{F}$, as well as the other arguments included in its definition in Section 3.2. (For this reason, our construction requires each dealer to have a distinct set of parameters for its sharings, i.e., produced by its own call to vssInit*.) This enables the dealer to evaluate $\mathcal{F}$ itself, which it does on $\langle r, i\rangle$ for each $i \in [n]$ (lines 10–12), where r is a new, random κ-bit nonce (line 9). The resulting values $\{y_i\}_{i\in[n]}$ are divided into ℓ groups of size $k-1$, each group being used to construct a set of $k-1$ points $\mathsf{Points}_j \leftarrow \{\langle i, y_i\rangle \mid (j-1)(k-1) < i \leq j(k-1)\}$ (line 14) on which vssMakeSecret is invoked (line 15). The resulting $s_j \in \mathbb{Z}_q[x]$ is then shared using $\mathcal{V}$ (line 16). Recall that by the definition of vssMakeSecret, each $u_i^*[j]$ thus produced satisfies $u_i^*[j] = y_i$. Of course, the input secret s is also shared (line 17). The results of these sharings are grouped according to replica index i and returned as u_i^* for each $i \in [n]$, along with all of the sharing commitments c^* and the nonce r (line 18).

vssVerify* and vssReconstruct* operate in the natural way. vssVerify* verifies the commitment $c^*[0]$ and share $u_i^*[0]$ (line 21) produced in the sharing of s, as well as verifying the commitment $c^*[j]$ and share $u_i^*[j]$ (line 28) produced in the sharing of s_j. In addition, it verifies (intuitively) that $u_i^*[j] = y_i$ (line 25). The latter two verifications are skipped if $u_i^*[1] = \perp$ (line 23), which occurs if the share u_i^* was recovered (see below). In this case, $u_i^*[j] = \perp$ for all $j \in [\ell]$ (or should be, and so any $j \in [\ell]$ for which $u_i^*[j] \neq \perp$ is just ignored). vssReconstruct* simply uses vssVerify* to verify each share u_i^* provided as input (line 33) and then submits $c^*[0]$ and the inputs $\{\langle vpk_i, u_i^*[0]\rangle\}_{i\in I}$ to vssReconstruct to reconstruct s (line 36).

$\mathsf{vssRecoverContrib}^*(c^*, r, vsk_i^*, u_i^*, \hat{\imath})$ is invoked at replica i to construct its contribution to enable replica $\hat{\imath}$ to reconstruct its share $u_{\hat{\imath}}^*$. vssRecoverContrib* returns $u_i^*[0]$ blinded by $u_i^*[j]$ (line 41) where $j \leftarrow \lceil \hat{\imath}/(k-1)\rceil$. Then, so that replica $\hat{\imath}$ can recover its share of the original secret, replica i also returns its share of the DPRF scheme $\mathcal{F}$ evaluated at $\langle r, \hat{\imath}\rangle$ (line 39).

$\mathsf{vssRecoverVerify}^*(c^*, r, v_i^*, vpk_i^*, \hat{\imath})$ is executed by replica $\hat{\imath}$ to verify that replica i performed vssRecoverContrib* correctly. The output of vssRecoverContrib* contributed by replica i is passed into vssRecoverVerify* as v_i^* and is parsed into its constituent components in line 44. First, the DPRF contribution d_i is checked on line 46 to ensure that it corresponds to a correct evaluation of the DPRF scheme $\mathcal{F}$ at the point $\langle r, \hat{\imath}\rangle$. vssRecoverVerify* then combines the commitments (line 48) and uses vssVerify (line 49) to check that the blinded share u was created correctly. If both checks pass, then vssRecoverVerify* returns true.

$\mathsf{vssRecover}^*(c^*, r, \{\langle vpk_i^*, v_i^*\rangle\}_{i\in I}, \hat{\imath}, vpk_{\hat{\imath}}^*)$ is executed at replica $\hat{\imath}$ to recover its share $u_{\hat{\imath}}^*$. In particular, $u_{\hat{\imath}}^*[0]$ will be a share of the original polynomial for $\hat{\imath}$. vssRecover* first invokes vssRecoverVerify* to make sure that the share sent by each replica $i \in I$ is correct

```
1:  procedure vssInit*$(1^\kappa, k, n)$
2:      $\langle q, \{\langle vpk_i, vsk_i\rangle\}_{i\in[n]}\rangle \leftarrow$ vssInit$(1^\kappa, k, n)$
3:      $\langle\{\langle dpk_i, dsk_i\rangle\}_{i\in[n]}\rangle \leftarrow$ dprfInit$(1^\kappa, k, n, \{0,1\}^\kappa \times [n], \mathbb{Z}_q)$
4:      for $i \in [n]$ do
5:          $vpk_i^* \leftarrow \langle vpk_i, dpk_i\rangle$
6:          $vsk_i^* \leftarrow \langle vsk_i, dsk_i\rangle$
7:      return $\langle q, \{\langle vpk_i^*, vsk_i^*\rangle\}_{i\in[n]}\rangle$

8:  procedure vssShare*$(s, q, \{dsk_i\}_{i\in[n]}, \{vpk_i^*\}_{i\in[n]})$
9:      $r \xleftarrow{\$} \{0,1\}^\kappa$
10:     for $i \in [n]$ do
11:         $\langle vpk_i, dpk_i\rangle \leftarrow vpk_i^*$
12:         $y_i \leftarrow$ dprfEval$(\langle r, i\rangle, \{$dprfContrib$(dsk_{\hat{i}}, \langle r, i\rangle)\}_{\hat{i}\in[n]})$
13:     for $j \in [\ell]$ do
14:         Points$_j \leftarrow \{\langle i, y_i\rangle \mid (j-1)(k-1) < i \le j(k-1)\}$
15:         $s_j \leftarrow$ vssMakeSecret$(q,$ Points$_j)$
16:         $\langle c^*[j], \{u_i^*[j]\}_{i\in[n]}\rangle \leftarrow$ vssShare$(s_j, q, \{vpk_i\}_{i\in[n]})$
17:     $\langle c^*[0], \{u_i^*[0]\}_{i\in[n]}\rangle \leftarrow$ vssShare$(s, q, \{vpk_i\}_{i\in[n]})$
18:     return $\langle c^*, r, \{u_i^*\}_{i\in[n]}\rangle$

19: procedure vssVerify*$(vpk_i^*, c^*, r, u_i^*)$
20:     $\langle vpk_i, dpk_i\rangle \leftarrow vpk_i^*$
21:     if vssVerify$(vpk_i, c^*[0], u_i^*[0])$ = false then
22:         return false
23:     if $u_i^*[1] \ne \perp$ then
24:         $j \leftarrow \lceil i/(k-1)\rceil$
25:         if dprfVerify$(dpk_i, \langle r, i\rangle, u_i^*[j])$ = false then
26:             return false
27:         for $j \in [\ell]$ do
28:             if vssVerify$(vpk_i, c^*[j], u_i^*[j])$ = false then
29:                 return false
30:     return true

31: procedure vssReconstruct*$(c^*, r, \{\langle vpk_i^*, u_i^*\rangle\}_{i\in I})$
32:     for $i \in I$ do
33:         if vssVerify*$(vpk_i^*, c^*, r, u_i^*)$ = false then
34:             return $\perp$
35:         $\langle vpk_i, dpk_i\rangle \leftarrow vpk_i^*$
36:     return vssReconstruct$(c^*[0], \{\langle vpk_i, u_i^*[0]\rangle\}_{i\in I})$

37: procedure vssRecoverContrib*$(c^*, r, vsk_i^*, u_i^*, \hat{i})$
38:     $\langle vsk_i, dsk_i\rangle \leftarrow vsk_i^*$
39:     $d_i \leftarrow$ dprfContrib$(dsk_i, \langle r, \hat{i}\rangle)$
40:     $j \leftarrow \lceil \hat{i}/(k-1)\rceil$
41:     return $\langle d_i, (u_i^*[0] + u_i^*[j])\rangle$

42: procedure vssRecoverVerify*$(c^*, r, v_i^*, vpk_i^*, \hat{i})$
43:     $j \leftarrow \lceil \hat{i}/(k-1)\rceil$
44:     $\langle d_i, u\rangle \leftarrow v_i^*$
45:     $\langle vpk_i, dpk_i\rangle \leftarrow vpk_i^*$
46:     if dprfVerify$(dpk_i, \langle r, \hat{i}\rangle, d_i)$ = false then
47:         return false
48:     $c \leftarrow$ vssCombineCommitments$(c^*[0], c^*[j])$
49:     if vssVerify$(vpk_i, c, u)$ = false then
50:         return false
51:     return true

52: procedure vssRecover*$(c^*, r, \{\langle vpk_i^*, v_i^*\rangle\}_{i\in I}, \hat{i}, vpk_{\hat{i}}^*)$
53:     $j \leftarrow \lceil \hat{i}/(k-1)\rceil$
54:     for $i \in I$ do
55:         if vssRecoverVerify*$(c^*, r, v_i^*, vpk_i^*, \hat{i})$ = false then
56:             return $\perp$
57:         $\langle d_i, u_i\rangle \leftarrow v_i^*$
58:         $\langle vpk_i, dpk_i\rangle \leftarrow vpk_i^*$
59:     $c \leftarrow$ vssCombineCommitments$(c^*[0], c^*[j])$
60:     $s \leftarrow$ vssReconstruct$(c, \{\langle vpk_i, u_i\rangle\}_{i\in I})$
61:     $y_{\hat{i}} \leftarrow$ dprfEval$(\langle r, \hat{i}\rangle, \{d_i\}_{i\in I})$
62:     $u_{\hat{i}}^* \leftarrow \langle (s(\hat{i}) - y_{\hat{i}}), \perp, \ldots, \perp\rangle$
63:     if vssVerify*$(vpk_{\hat{i}}^*, c^*, r, u_{\hat{i}}^*)$ = false then
64:         return $\perp$
65:     return $u_{\hat{i}}^*$
```

Figure 1: Pseudocode for our VSS scheme

(line 55). vssRecover* then leverages vssReconstruct (line 60) to reconstruct a polynomial $s \in \mathbb{Z}_q[x]$ that is the sum of the polynomial originally shared in vssShare* that resulted in commitment $c^*[0]$ and the j-th masking polynomial s_j that resulted in commitment $c^*[j]$, where $j = \lceil \hat{i}/(k-1)\rceil$. vssRecover* then evaluates $s(\hat{i})$ and subtracts $s_j(\hat{i}) = \text{dprfEval}(\langle r, \hat{i}\rangle, \{d_i\}_{i\in I})$ (lines 60–62) to obtain $u_{\hat{i}}^*[0]$.

3.6 Security

Below, we sketch a proof that our modified VSS scheme preserves the security properties from the underlying VSS protocol.

Hiding First note that if a hiding adversary $\mathcal{A}_{\mathcal{V}^*}$ for $\mathcal{V}^*$ is legitimate, then it is legitimate for both $\mathcal{F}$ and $\mathcal{V}$. Consider an execution in which $O_{\mathcal{F}}^{?}$ (used in place of dprfEval) is instantiated as $O_{\mathcal{F}}^{\text{rand}}$. For any commitment c^*, the set of indices I for which $\mathcal{A}_{\mathcal{V}^*}$ obtains the shares $\{u_i^*[0]\}_{i\in I}$ produced in line 17 (i.e., in its invocation of $O_{\mathcal{V}^*}^{b}$ that returned c^*) satisfies $|I| < k$. To see why, note that $\mathcal{A}_{\mathcal{V}^*}$ can obtain $u_i^*[0]$ for any i in one of three ways: (i) by invoking $O_{\mathcal{V}^*,i}.compromise$; (ii) by invoking $O_{\mathcal{V}^*,i}.contrib(c^*)$; or (iii) by invoking $O_{\mathcal{V}^*,\hat{i}}.recover(c^*, i)$ at each $\hat{i} \in \hat{I}$ where $|\hat{I}| \ge k$, in which case $\mathcal{A}_{\mathcal{V}^*}$ can recover $u_i^*[0]$ using the vssRecover* routine (line 62). Critically, invoking $O_{\mathcal{V}^*,\hat{i}}.recover(c^*, i)$ at each $\hat{i} \in \hat{I}$ where $|\hat{I}| < k$ yields no information about $u_i^*[0]$, since when $|\hat{I}| < k$, the value y_i is random (line 61) and so $u_i^*[0] = s(i) - y_i$ (line 62) is hidden information-theoretically. Because $\mathcal{A}_{\mathcal{V}^*}$ is legitimate, it thus obtains $u_i^*[0]$ for only fewer than k values of i, and so if its success (in the sense of (2)) is nonnegligible in κ, then we can construct a hiding attacker for $\mathcal{V}$ with success nonnegligible in κ, as well.

Now suppose $\mathcal{A}_{\mathcal{V}^*}$ has success in the execution above that is only negligible in κ, and consider an execution in which $O^{?}_{\mathcal{F}}$ is instead instantiated as $O^{\text{real}}_{\mathcal{F}}$. If $\mathcal{A}_{\mathcal{V}^*}$ now has success that is nonnegligible in κ, then we can use $\mathcal{A}_{\mathcal{V}^*}$ to construct a DPRF attacker for $\mathcal{F}$ with success nonnegligible in κ (in the sense of (1)).

To summarize, if there is a hiding attacker that breaks $\mathcal{V}^*$ with nonnegligible probability, then either there is a hiding attacker that breaks $\mathcal{V}$ with nonnegligible probability or there is a DPRF attacker that breaks $\mathcal{F}$ with nonnegligible probability.

Binding A binding adversary $\mathcal{A}_{\mathcal{V}^*}$ is provided inputs $\langle q, \{\langle vpk^*_i, vsk^*_i\rangle\}_{i\in[n]}\rangle \leftarrow \mathsf{vssInit}^*(1^\kappa, k, n)$, and *succeeds* if it outputs c^*, $\{u^*_i\}_{i\in I}$ and $\{\hat{u}^*_i\}_{i\in\hat{I}}$ for which

$$\begin{array}{ll} & \mathsf{vssReconstruct}^*(c^*, r, \{\langle vpk^*_i, u^*_i\rangle\}_{i\in I}) = s \\ \wedge & \mathsf{vssReconstruct}^*(c^*, r, \{\langle vpk^*_i, \hat{u}^*_i\rangle\}_{i\in \hat{I}}) = \hat{s} \\ \wedge & s \neq \perp \wedge \hat{s} \neq \perp \wedge s \neq \hat{s} \end{array}$$

Let s and $\hat{s}$ be values satisfying this condition. Then,

$$s = \mathsf{vssReconstruct}(c^*[0], \{\langle vpk_i, u^*_i[0]\rangle\}_{i\in I})$$
$$\hat{s} = \mathsf{vssReconstruct}(c^*[0], \{\langle vpk_i, u^*_i[0]\rangle\}_{i\in \hat{I}})$$

where $\langle q, \{\langle vpk_i, vsk_i\rangle\}_{i\in[n]}\rangle \leftarrow \mathsf{vssInit}(1^\kappa, k, n)$ (see lines 2 and 36). That is, breaking binding for $\mathcal{V}^*$ implies breaking binding for $\mathcal{V}$, and so if $\mathcal{V}$ ensures the binding property, then so does $\mathcal{V}^*$.

3.7 Instantiating VSSR

Now we discuss how to instantiate VSSR concretely given a DPRF scheme [38] and a VSS scheme from Kate et al. [29]. We also instantiate our VSSR with a VSS scheme from Pedersen [42], whose details can be found in Appendix B.

3.7.1 Kate et al. Secret Sharing We describe how to fit the secret sharing scheme from Kate et al. [29] into our framework. Note that this secret sharing scheme also has a *witness*, which proves that a particular share is consistent with the polynomial commitment. Witnesses are additively homomorphic as well and can be manipulated the same way as the shares can. In particular, we can perform polynomial interpolation in order to take a set of f witnesses and obtain the witness for any other share. Additionally, we only need to send a witness when we transmit the corresponding share. Thus, witnesses only increase the communication overhead by a constant factor. In the description below, we assume that we have the witness corresponding to each share.

- $\mathsf{vssInit}(1^\kappa, k, n)$ first chooses a safe prime q at least κ bits in length. Then, we initialize two groups of order q: $\mathbb{G}$ and $\mathbb{G}_t$ such that there exists a bilinear map $e\colon \mathbb{G}\times\mathbb{G} \longrightarrow \mathbb{G}_t$. We then generate a $\tau \in \mathbb{Z}_q$ and pick a generator $g \in \mathbb{G}$. Set vpk_i to be $\langle \mathbb{G}, \mathbb{G}_t, e, g, \{g^{\tau^j}\}_{j\in[f]}\rangle$ and vsk_i to be $\perp$ for all i. Then, we delete τ. Finally, $\mathsf{vssInit}$ returns $\langle q, \{\langle vpk_i, vsk_i\rangle\}_{i\in[n]}\rangle$.
- $\mathsf{vssShare}(s, q, \{vpk_i\}_{i\in[n]})$ first extracts the public key and gets g and $\{g^{\tau^j}\}_{j\in[f]}$. Let s_j be the coefficient of the x^j term in s and $s(i)$ be the evaluation of s at point i. We now compute $g^{s(\tau)}$ by computing $\prod_{j=0}^{f}(g^{\tau^j})^{s_j}$ and assign it to c. Now, using polynomial division, we can compute the coefficients of $\frac{s(x)-s(i)}{x-i}$, which will allow us to compute $g^{\frac{s(\tau)-s(i)}{\tau-i}}$ which is the witness for u_i. We also set u_i to be $s(i)$. Finally, $\mathsf{vssShare}$ returns $\langle c, \{u_i\}_{i\in[n]}\rangle$.
- $\mathsf{vssVerify}(vpk_i, c, u_i)$ first extracts $g^{s(\tau)}$ from c, $s(i)$ from u_i, and e, g, g^τ from vpk_i. We also have access to the value $g^{\frac{s(\tau)-s(i)}{\tau-i}}$ since the witness for the share is transmitted along with the share. Then, $\mathsf{vssVerify}$ returns true if $e(g^{s(\tau)}, g)$ equals $e(g^{\frac{s(\tau)-s(i)}{\tau-i}}, \frac{g^\tau}{g^i})e(g,g)^{s(i)}$ and false otherwise.
- $\mathsf{vssReconstruct}(c, \{\langle vpk_i, u_i\rangle\}_{i\in I})$ first calls $\mathsf{vssVerify}(vpk_i, c, u_i)$ for all $i \in I$. If all of $\mathsf{vssVerify}$ calls return true, then we continue. Otherwise, $\mathsf{vssReconstruct}$ returns $\perp$. Then, we extract $s(i)$ from each u_i and do Lagrange interpolation to identify the original polynomial and return that value.
- $\mathsf{vssMakeSecret}(q, \{\langle x_i, y_i\rangle\}_{i\in I})$ does a Lagrange interpolation in order to identify the unique degree $k-1$ polynomial in $\mathbb{Z}_q[x]$ that goes through (x_i, y_i) and returns that as s.
- $\mathsf{vssCombineCommitments}(c, \hat{c})$ first extracts $g^{s(\tau)}$ from c and $g^{\hat{s}(\tau)}$ from $\hat{c}$. We then set $\check{c}$ to $(g^{s_\tau})(g^{\hat{s}(\tau)})$ and return that value.

3.7.2 DPRF Instantiation Our distributed pseudorandom function $\mathcal{F}$ consists of four algorithms: $\mathsf{dprfInit}$, $\mathsf{dprfContrib}$, $\mathsf{dprfVerify}$, and $\mathsf{dprfEval}$. Our implementation defines them as follows [38]:

- $\mathsf{dprfInit}(1^\kappa, k, n, \mathbf{D}, \mathbb{Z}_q)$, first chooses a generator h of $\mathbb{G}$ of order q. A k out of n secret sharing of a private value $\alpha \in \mathbb{Z}_q$ is produced using Shamir secret sharing [45], of which the shares are $\{\alpha_i\}_{i\in[n]}$. $dpk_{\hat{i}}$ is set to $\langle h, h^\alpha, \{h^{\alpha_i}\}_{i\in[n]}\rangle$ for all $\hat{i} = 1..n$. $dsk_{\hat{i}}$ is set to $\alpha_{\hat{i}}$ for all $\hat{i} = 1..n$. $\mathsf{dprfInit}$ outputs $\{\langle dpk_i, dsk_i\rangle\}_{i\in[n]}$.
- $\mathsf{dprfContrib}(dsk_i, x)$ first computes $f_i(x) = H(x)^{\alpha_i}$ where $H : \{0,1\}^* \longrightarrow \mathbb{G}$ is a hash function that is modeled as a random oracle. Here, α_i is obtained from the dsk_i. Let r be a randomly generated element of $\mathbb{Z}_{q'}$. Then, we let $c_i \leftarrow H'(H(x), h, f_i(x), h^{\alpha_i}, H(x)^r, h^r)$, where $H' : \{0,1\}^* \longrightarrow \mathbb{Z}_q$ is a hash function modeled as a random oracle. We set $z_i \leftarrow \alpha_i c_i + r \bmod q$. $\mathsf{dprfContrib}$ then outputs $\langle f_i(x), z_i, c_i\rangle$.
- $\mathsf{dprfVerify}(dpk_i, x, d)$ first extracts $f_i(x)$, c_i, and z_i from d. Then, h and h^{α_i} are extracted from dpk_i. Finally, $\mathsf{dprfVerify}$ returns true if $c_i = H'(H(x), h, f_i(x), h^{\alpha_i}, H(x)^{z_i} f_i(x)^{-c_i}, h^{z_i}(h^{\alpha_i})^{-c_i})$.
- $\mathsf{dprfEval}(x, \{d_i\}_{i\in I})$ first verifies each d_i using $\mathsf{dprfVerify}$. If any of the verifications returns false, then $\mathsf{dprfEval}$ returns $\perp$. Otherwise, we extract $f_i(x)$ values from each d_i. Since the exponents of $f_i(x)$ were shared in the exponent using Shamir secret sharing, $\mathsf{dprfEval}$ uses Lagrange interpolation in the exponent to get the value of $\mathcal{F}$ at x, hashes it into an element of $\mathbb{Z}_q$ and outputs that value.

4 Secret Shared State on PBFT

In this section we describe how to build a private replicated key/value store with Byzantine Fault Tolerance by incorporating VSSR into PBFT [12]. The service provides two APIs, put(K, V) and get(K) by which a client can write a value to a key or read a value previously written to a key.

Our design assumes a partial asynchronous Byzantine model. Specifically, we have $n = 3f+1$ replicas, $\leq f$ of which are Byzantine. The correctness of the value read from a key is ensured despite up to f Byzantine-faulty replicas, and values are also kept private from f faulty replicas, using our verifiable secret-sharing approach. Similar to previous works (e.g., [9, 37]), a client shares a secret value directly among the replicas, and a consensus protocol drives agreement on a verifiable digest of the value.

The network is assumed to be asynchronous, but will eventually go through periods of synchrony in which messages are delivered within a known time bound and correct replicas and clients make progress at a known rate. The network assumption is due to PBFT's [12] network assumptions. VSSR's construction is completely independent of the network model of the underlying BFT protocol. We assume that each message is signed by its sender so that its origin is known, subject to standard cryptographic assumptions.

Every client in the system is allowed to view all keys in the store. However, the service maintains a (potentially dynamic) access control policy that specifies which values each client can open. Under these assumptions, we provide the standard guarantees provided by a Byzantine fault tolerant protocol:

- **Linearizability** [26]. If a client sends a request to the replicated service, then the service's response is consistent with an execution where the client's request was executed instantaneously at some point between when the request was sent and the response was received.
- **Liveness**. If the network is synchronous, then every client request will get a response.

In addition to these standard properties, our design offers the following privacy property:

- **Privacy**. A value written to a key by a correct client where the access-control policy prohibits access by any faulty client, remains hidden from f Byzantine servers.

4.1 Setup

In addition to setting up authenticated communication channels among all parties in a setup phase, vssInit* is called for every client in the system and is part of the public/private key infrastructure. The client takes the role of the dealer in vssInit* while each replica takes the role of a participant. In particular, each client knows the secret keys for all replicas returned from its invocation of vssInit*.

Every replica stores a full copy of the K-V store. For each key there are two value entries, a public value (keyed K-pub) and a private value (keyed K-priv). A replica maintains a bounded log of pending commands that cannot grow beyond a certain system-wide parameter W. Once a command in the log is committed by the system, it is applied to the K-V store.

Views. Our solution employs a classical framework [12, 19] that revolves around an explicit ranking among proposals via *view* numbers. Replicas all start with an initial view, and progress from one view to the next. They accept requests and respond to messages only in their current view. In each view there is a single designated *leader.* In a view, zero or more decisions may be reached. If a sufficient number of replicas suspect that the leader is faulty, then a view change occurs and a new leader is elected. The precise way views are changed are described in Appendix C.

4.2 Common Mode Protocol

A client put is split into two parts, public and private. More specifically, in a put(K, V) request, the client privately shares V via vssShare*, and sends each share to its corresponding replica. The public part of put(K, V) consists of a client sending a put(K, c_V) request to the current leader. c_V is a global commitment to the polynomial s that binds the share of each replica as a verifiable share of s.

The leader waits until its local log has length $< W$. It then extends its local log with the put request, and sends a pre-prepare (ordering-request) containing its log tail.

A replica *accepts* a pre-prepare from the leader of the current view if it is well-formatted, if it extends any previous pre-prepare from this leader, if its log has fewer than W pending entries, and if the replica received a valid share corresponding to c_V. If the leader pre-prepare message has a valid format, but the replica did not receive the corresponding share for it, it starts a timer for share-recovery (see Appendix C).

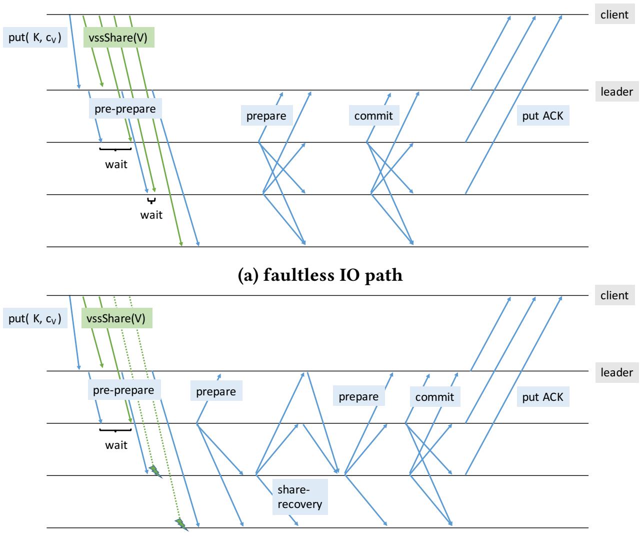
After accepting the pre-prepare, a replica follows the regular PBFT protocol. The replica first extends its local log to include the new request and broadcasts a *prepare* message to all replicas that includes the new log tail. Replicas wait to collect a *commit-certificate*, a set of $2f + 1$ prepare responses for the current log tail. Then the replica broadcasts a *commit* message carrying the commit-certificate to the other replicas. A decision is reached in a view on a new log tail when $2f + 1$ distinct replica have sent a commit message for it.

When a replica learns that a put(K, V) request has been committed to the log, it inserts to its local key-value store two entries, a global entry (K-pub, c_V) containing the global commitment to V, and a private entry (K-priv, u_i^*) containing the replica's private share u_i^*. The replica then responds to the client with a put acknowledgement message containing K and c_V. A client waits to receive $2f + 1$ put responses to complete the request. Figure 2a depicts the put io path, and Figure 2b the put io path when shares are missed.

The client get(K) protocol consists of sending the get request to the current leader. The pre-prepare, prepare and commit phases of the ordering protocol are carried as above, without the need to wait for shares. At the final stage, when a replica executes the get requests, it returns its share to the client in a response. If the replica is missing its share, it initiates the share-recovery protocol. The client waits to receive $f + 1$ valid get responses. It uses vssVerify* to verify each response, and vssReconstruct* to reconstruct the secret value from the responses.

4.3 Share-Recovery Protocol

There are several circumstances in the protocol when a replica discovers it is missing its private share of a request and needs to recover it. To initiate share-recovery, a replica broadcasts a recovery request. Other replicas respond to a share-recovery request with the output of vssRecoverContrib*. After receiving a response, the original replica uses vssRecoverVerify* to check the response. If the response is valid, then it is stored, and if it is invalid, then it is dropped. When it receives $f + 1$ valid responses, the replica uses vssRecover* to recover its missing secret share.



(a) faultless IO path

(b) IO path with recovery

Figure 2: put **common mode**

4.4 Common mode performance

The common mode protocol incurs the following performance costs. The client interaction with the BFT replicated service is linear, since it needs to populate all replicas with shares. Additionally, the client collects $f + 1$ responses from servers.

The communication among the replicas to achieve an ordering decision is quadratic. There are several practical variants of BFT replication that achieve linear communication during periods of synchrony and when a leader is non-faulty (e.g., [31, 35**?**]) These improvements are left outside the scope of this paper. However, our modified VSS protocol is designed so it can be incorporated within them without increasing the asymptotic complexity of the common mode.

In terms of latency, the sharing protocol is non-interactive and single-round, and so it can be performed concurrently with the leader broadcast. Recovery incurs extra latency since each replica must ask at least $f+1$ correct replicas for their contributions. In the original BFT protocol, recovering a missing request only requires asking 1 correct replica for the request data. In both cases, the recovery protocol is interactive and single-round, so there are no asymptotic increases in latency. However, in practice, there will be a difference in latency between the two scenarios.

Proof Intuition The full proof that our construction satisfies the linearizability, liveness and privacy properties above is in Appendix A. We present high level intuition behind our proofs here.

To show linearizability and liveness, we show that every execution of our secret shared PBFT protocol can be mapped to an execution of an unmodified PBFT protocol. Since the original PBFT protocol satisfies linearizability and liveness, so does our modified protocol. Privacy is shown by using the fact that to recover a secret shared value, an adversary must obtain the cooperation of at least one correct replica.

5 Implementation

We implement a secret shared BFT engine by layering PBFT [12] with our secret sharing scheme. Our implementation consists of 4700 lines of Python and 4800 lines of C. We optimize our design for multi-core environments, with one network thread running on a core which never blocks. Additionally, we use one thread for every other core in order to do all cryptographic operations that are required by PBFT and our secret sharing scheme. We use elliptic curve signatures with the secp256k1 library for all signature checking operations and the Relic library [2] for all other cryptographic operations related to our scheme. We also make a few optimizations for the Kate et al. and Pedersen secret sharing schemes in order to make them faster.

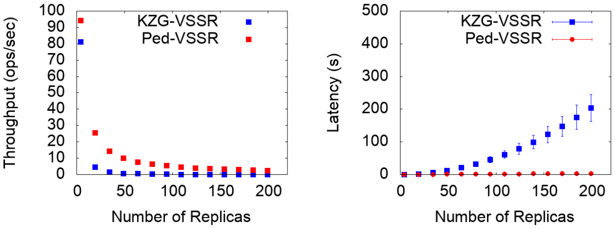
Kate et al. Kate et al.'s secret sharing scheme lends itself for extensive caching during setup time. Once the powers g^{τ^j} are known for all j, we construct precomputation tables for each coefficient so that all exponentiations during runtime leverage these tables for efficiency. In the sharing step, we first use the well known Horner's method to optimize the share evaluation. However, we also note that each intermediate value obtained in Horner's method when evaluating $s(i)$ is also the coefficient of the quotient polynomial $\frac{s(x)-s(i)}{x-i}$ which means that we can do the necessary division required for free before using our precomputation tables to evaluate the quotient at τ. In the share verification step, every verification requires the value of $e(g, g)$ so we can precompute that as well to save a bilinear map operation. Also in the share verification phase, the division of $\frac{g^\tau}{g^i}$ only has n possible values, which means that we can precompute all of these values as well. Finally, when doing Lagrange interpolation, we know that the indices range from 0 to $n - 1$ and in the denominator, we need to compute the product of differences of these indices. Thus, to avoid taking inverses, we simply take inverses of all n values of the differences which means that during runtime, we only have to do multiplications.

Pedersen Pedersen's secret sharing scheme does not lend itself to as much caching since most of the values are unknown beforehand. However, we do generate precomputation tables for both g and h during setup and compute the inverses to make Lagrange interpolation easier.

6 Evaluation

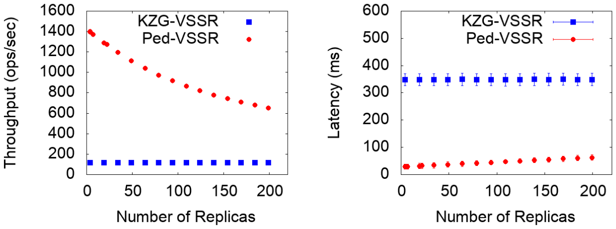
Our evaluation seeks to answer two basic questions. First, we investigate the costs of each API call in our secret sharing scheme. Then, we look at how expensive it is to incorporate our secret sharing scheme into a BFT key value store. We instantiate VSSR using the DPRF in Naor et al. [38] and two different VSS schemes: Pedersen's VSS scheme [42] and Kate et al.'s VSS scheme [29]. We call the Pedersen instantiation Ped-VSSR and the Kate et al. instantiation KZG-VSSR. Our implementation uses the Relic [2] cryptographic library and, for our elliptic-curve algorithms, the BN_P254 curve.

We build a private BFT key value store using PBFT replication [12], implemented in Python and C, incorporating VSSR into the write path of the algorithm.



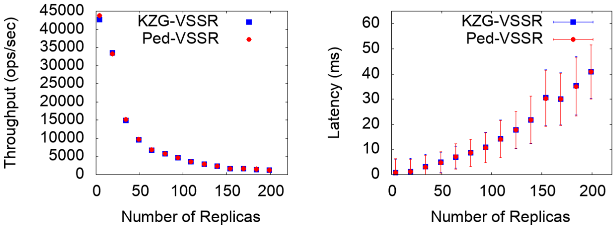
(a) vssShare* **throughput** **(b)** vssShare* **latency**

Figure 3: vssShare* **latency and throughput vs.** n



(a) vssVerify* **throughput** **(b)** vssVerify* **latency**

Figure 5: vssVerify* **latency and throughput vs.** n



(a) vssReconstruct* **throughput** **(b)** vssReconstruct* **latency**

Figure 6: vssReconstruct* **latency and throughput vs.** n

6.1 Microbenchmarks

For our microbenchmarks, we evaluate each function in our VSSR scheme. We vary the number of replicas from 4 to 211 and measure the latency and throughput of each operation. We use EC2 `c5.xlarge` instances in order to run our microbenchmarks, which have 4 virtual CPUs per instance.

The module that implements our secret sharing scheme optimizes for throughput, while compromising slightly on latency. Each API call runs on a single core; the task is run to completion and the result is returned in the order that the tasks were enqueued. This maximizes for throughput due to the lack of cross core communication, but at the expense of request latency as many of the underlying cryptographic operations can leverage multi-core environments to execute faster.

Each microbenchmark ran for at least 60 seconds and collected at least 30 samples. Before computing the final statistic, we ignored any requests that were completed in the first 10 seconds and the last 10 seconds of the run. We report the aggregate throughput during the run and the mean and standard deviation of the latency of each request completed in our run.

6.1.1 vssShare* *Microbenchmark* Figure 3a shows that Ped-VSSR can sustain more sharings per second than KZG-VSSR for all cluster sizes. This gap increases as the number of replicas increases. The difference between Ped-VSSR and KZG-VSSR is due to the underlying VSS scheme. KZG-VSSR computes witnesses for each share, which involves evaluating a polynomial in the elliptic curve group. Additionally, the throughput decrease is quadratic since evaluating each share (or witness) takes $O(n)$ CPU time and there are n shares. So, vssShare* takes $O(n^2)$ time for both KZG-VSSR and Ped-VSSR. We see this cost mirrored in the latency graph as well in Figure 3b.

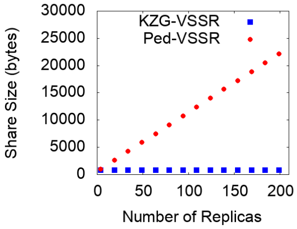


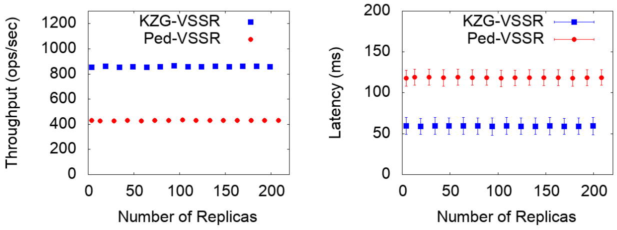
Figure 4: Volume the dealer transmits per replica to share a secret among n **replicas**

Figure 4 shows the size of the share and associated metadata that is sent to each replica when the client shares a single 254 bit integer, which is equal to the disk space that the replica needs to store a secret shared value. As expected, Ped-VSSR has a linearly increasing bandwidth and storage footprint with respect to the cluster size. Meanwhile, KZG-VSSR only requires each replica to store 860 bytes of information irrespective of the cluster size.

6.1.2 vssVerify* *Microbenchmark* Figure 5 shows that the throughput and latency of verifying a share, which is done by the replicas upon receiving a share. We see that KZG-VSSR's latency and throughput stays constant at 117 operations per second with a 350 millisecond mean latency irrespective of the cluster size. Meanwhile, Ped-VSSR's throughput decreases and latency increases as the number of replicas in the cluster increases. We see that KZG-VSSR is asymptotically faster than Ped-VSSR, but Ped-VSSR's cheaper cryptographic operations still causes it to outperform KZG-VSSR.

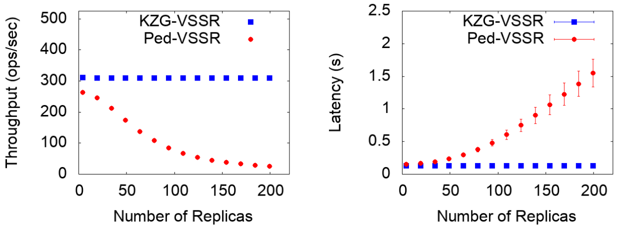
6.1.3 vssReconstruct* *Microbenchmark* vssReconstruct* has almost identical performance between KZG-VSSR and Ped-VSSR, as we see in Figure 6. vssReconstruct* does not include the time taken to run vssVerify* since share verification happens when the message itself is verified. Figure 6a shows that vssReconstruct* can occur at very high throughput with its performance dropping off quadratically. Figure 6b similarly shows that the latency is increasing quadratically as the cluster size increases. This quadratic performance hit is due to the quadratic number of modular multiplications each vssReconstruct* requires.

6.1.4 vssRecoverContrib* *Microbenchmark* Figure 7 shows that vssRecoverContrib* throughput and latency is independent of the cluster size for both KZG-VSSR and Ped-VSSR. Additionally, Ped-VSSR's vssRecoverContrib* has exactly half the throughput (430 vs. 860) and twice the latency (118 ms vs. 59 ms) of KZG-VSSR. This is a side effect of using recovery polynomials to do share recovery. vssRecoverContrib* only has to do a constant amount of work per recovery. Ped-VSSR's vssRecoverContrib* operation requires two polynomial shares to be recovered while while KZG-VSSR only requires one.



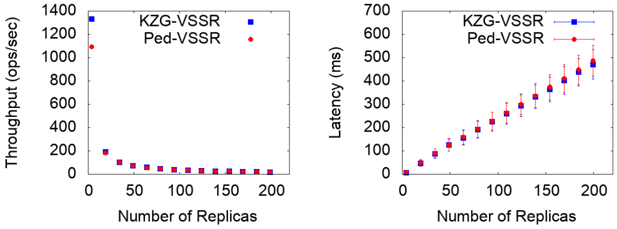
(a) vssRecoverContrib* **throughput** **(b)** vssRecoverContrib* **latency**

Figure 7: vssRecoverContrib* **latency and throughput vs.** *n*



(a) vssRecoverVerify* **throughput** **(b)** vssRecoverVerify* **latency**

Figure 8: vssRecoverVerify* **latency and throughput vs.** *n*

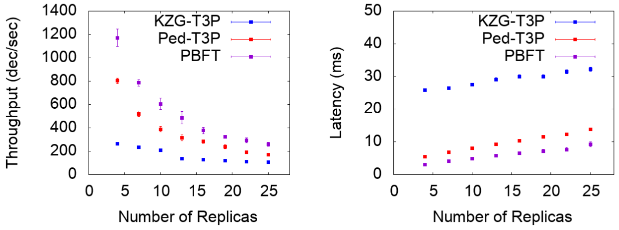


(a) vssRecover* **throughput** **(b)** vssRecover* **latency**

Figure 9: vssRecover* **latency and throughput vs.** *n*

6.1.5 vssRecoverVerify Microbenchmark* Figure 8 shows that KZG-VSSR's vssRecoverVerify* operation has higher throughput and lower latency than Ped-VSSR. This performance difference occurs since vssRecoverVerify* must combine commitments and witnesses from the contributions received from vssRecoverContrib*. KZG-VSSR performs this computation using a constant number of elliptic curve multiplications whereas Ped-VSSR computes this using a linear number of elliptic curve multiplications. Thus, as the cluster size increases, Ped-VSSR's performance also degrades accordingly.

6.1.6 vssRecover Microbenchmark* Similar to vssReconstruct*, share verification via vssRecoverVerify* happens in our implementation upon receiving each share from a replica. Figure 9 shows that vssRecover* incurs costs primarily due to interpolation (like vssReconstruct*), evaluation of the DPRF and interpolation of any witnesses. Therefore, asymptotically, we see in Figure 9a and Figure 9b that vssRecover* behaves similarly to vssReconstruct* but with an order of magnitude lower throughput and an order of magnitude higher latency.



(a) PUT operation throughput **(b) PUT operation latency**

Figure 10: PUT operation latency and throughput vs. *n*

6.2 Incorporating VSSR into PBFT

We incorporate VSSR into a PBFT implementation in order to implement a threshold trusted third party (T3P). We instantiate our T3P using KZG-VSSR and Ped-VSSR, which we will refer to as KZG-T3P and Ped-T3P. We also implement and evaluate a key-value store on top of KZG-T3P, Ped-T3P, and PBFT.

To generate load in our evaluation, a client sends PUT requests asynchronously to the primary. The client pregenerates the requests to send to the cluster and loops through them once they are finished. For our throughput experiments, the clients asynchronously send enough requests at a time to saturate the system. For our latency benchmarks, the clients send requests serially and measure the latency of each request. We used Amazon AWS to run our tests and used `c5.4xlarge` instances for all clients and replicas.

Similar to our microbenchmarks, our implementation uses the Relic [2] cryptographic library for most cryptographic operations and the BN_P254 elliptic curve. For signatures in PBFT, our implementation uses the optimized `secp256k1` library used in Bitcoin.

6.2.1 Benchmarks Figure 10b shows that for all schemes and cluster sizes, the request latency is less than 35 milliseconds. We see the expected performance hits, such as KZG-T3P having lower performance than Ped-T3P and secret sharing having some overhead over vanilla PBFT. However, the total latencies show that our scheme is efficient enough to be used in certain scenarios.

Figure 10a shows the throughput overhead of secret sharing. Due to the quadratic overhead of Ped-T3P and PBFT, we see that the performance penalty stays roughly constant over all cluster sizes. Since KZG-T3P is linear, we see the performance overhead shrinking slightly as the cluster size increases. PBFT took a 78% performance hit in throughput when going from $n = 4$ to $n = 25$. Similarly, Ped-T3P took a 79% performance hit. However, KZG-T3P did a linear number of cryptographic operations that dominated the performance overhead and only suffered a 60% drop.

7 Related Work

This paper makes three primary contributions: VSSR, a framework for building VSS schemes that provides share recovery; KZG-VSSR, an instantiation of VSSR that has linear dealer cost on the sharing phase; and a threshold trusted third party (T3P) built by combining VSSR with a Byzantine Fault Tolerant state machine. We discuss the related works below.

7.1 Proactive secret sharing

A treatment of the prior work in verifiable secret sharing and asynchronous verifiable secret sharing is given in Section 2.2. Unlike those prior works though, another way to approach share recovery is through proactive secret sharing. Proactive secret sharing has been used in threshold signature schemes [14, 16] and in storage systems [6, 47]. Using proactive secret sharing for share recovery would require sending a random polynomial that has nothing in common with the original shared polynomial except for the share that the recovering replica is interested in.

Prior work in proactive secret sharing [28] is difficult to apply directly to the problem of share recovery, however. Some works [27, 28] assume a synchronous broadcast channel that delivers to all replicas instantaneously, which greatly simplifies the problem of agreeing on a random recovery polynomial. Other proactive secret sharing systems [6, 17, 21, 39, 40, 47] require replicas to reshare their secret shares and a new polynomial is formed through interpolation. Batching [4, 5] and parallelization [22] also have been explored in proactive secret sharing schemes, and while batching provides similar asymptotic guarantees to KZG-VSSR, it does so at a large latency cost. PVSS [48] does not make any such assumption and can be used in VSSR, but it suffers from an exponential setup cost in the number of faults it tolerates, making it unusable for tolerating more than a few faults. MPSS [44] uses a Byzantine agreement protocol in order to explicitly agree on the random recovery polynomial, which would add a few additional rounds to VSSR if used in share reconstruction. CHURP [34] uses bivariate polynomials and thus the share refresh protocol incurs a quadratic cost.

Although closest "in spirit" to proactive recovery schemes, VSSR addresses only the share phase. It is left for future work to see whether proactive share recovery can be expedited with techniques borrowing from VSSR.

7.2 Privacy in BFT

Methods to store data across n storage nodes in a way that ensures the privacy, integrity, and availability of the data despite up to k of these nodes being compromised is a theme that has been revisited numerous times in the last 30 years (e.g., [18, 23, 25, 33, 46]). The proposals in this vein of research often do not defend against the misbehavior of the data writers. In particular, a data writer might deploy data to the storage nodes in a way that makes data recovery impossible or ambiguous, in the sense that the data reconstructed depends on which correct nodes cooperate to do so. Protecting against corrupt data writers is one of the primary goals of *verifiable* secret sharing and its derivatives, for which we've surveyed the most directly related works in Section 2.

With the rise of blockchains supporting smart contracts, there has been a resurgence of activity in finding ways to add privacy guarantees to Byzantine fault-tolerant algorithms, and indeed this is one motivation behind our work. Another class of approaches to this problem uses zero knowledge proofs [36, 43] for privacy. These approaches provide a very strong guarantee where it is impossible for anyone (other than the data owner) to recover the sensitive data, but where anyone can validate that the data satisfies some prespecified properties. However, such systems only work for a limited set of applications, rather than general purpose state machines that we target here. Additionally, these systems do not have any control over the data itself; i.e., the sensitive data must be managed by the owner, which is not suitable for a large class of applications.

CALYPSO [30] resolves this through the use of a publicly verifiable secret sharing scheme, but they require two BFT clusters—one for access control and one for secret management. Thus, their protocol requires more replicas to operate. Additionally, CALYPSO requires the access-control policy to be specified ahead of time by the client, whereas a T3P can easily allow dynamic access-control policies.

A related line of work addresses the problem of causal ordering in Byzantine systems using secret sharing [**? ? ?**]. However, these systems use secret sharing to hide commands from the replicated service for fairness and consequently reveal the commands immediately. Thus, these techniques do not help with our key problem of share recovery.

8 Conclusion

This paper investigates how to incorporate verifiable secret sharing (VSS) into Byzantine fault tolerance (BFT) protocols. Solving the VSS with share recovery problem is a necessary first step towards that goal. Thus, we presented a framework, VSSR, and instantiated it using two distinct VSS schemes in order to solve the VSS with share recovery problem. We then incorporated these two instantiations into PBFT and evaluated them with a private key value store.

9 Acknowledgements

We would like to thank the anonymous reviewers for their feedback on our paper. Ethan Cecchetti and Dragos-Adrian (Adi) Seredinschi provided helpful discussion on various aspects of the paper. This work was supported by NSF grant DGE-1650441. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] T. Yurek A. Kate, A. Miller. 2019. Brief Note: Asynchronous Verifiable Secret Sharing with Optimal Resilience and Linear Amortized Overhead. *arXiv preprint arXiv:1902.06095* (2019).

[2] D. F. Aranha and C. P. L. Gouvêa. [n.d.]. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[3] M. Backes, A. Datta, and A. Kate. 2013. Asynchronous computational VSS with reduced communication complexity. In *Topics in Cryptology – CT-RSA 2013 (LNCS)*, Vol. 7779. 259–276.

[4] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. 2014. How to Withstand Mobile Virus Attacks, Revisited. In *ACM Conference on Principles of Distributed Computing (PODC) 2014*. 293–302.

[5] J. Baron, K. El Defrawy, J. Lampkins, and R. Ostrovsky. 2015. Communication-Optimal Proactive Secret Sharing for Dynamic Groups. In *International Conference on Applied Cryptography and Network Security (ACNS) 2015*. 23–41.

[6] K. D. Bowers, A. Juels, and A. Oprea. 2009. HAIL: A high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and Communications Security (CCS) 2009*. 187–198.

[7] G. Bracha and S. Toueg. 1985. Asynchronous Consensus and Broadcast Protocols. *J. ACM* 32, 4 (Oct. 1985), 824–840.

[8] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. 2002. Asynchronous verifiable secret sharing and proactive cryptosystems. In *9th ACM Conference on Computer and Communications Security (CCS)*.

[] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology – CRYPTO '01*. 524–541.

[9] C. Cachin, K. Kursawe, and V. Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18 (July 2005), 219–246. Issue 3.

[10] R. Canetti and T. Rabin. 1993. Fast Asynchronous Byzantine Agreement with Optimal Resilience. In *25thACM Symposium on Theory of Computing (STOC)*. 42–51.

[11] I. Cascudo and B. David. 2017. SCRAPE: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 537–556.

[12] M. Castro and B. Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* 20, 4 (2002).

[13] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. 1985. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th IEEE Symposium on Foundations of Computer Science (FOCS)*. 383–395.

[14] R. Cramer, R. Gennaro, and B. Schoenmakers. 1997. A Secure and Optimally Efficient Multi-Authority Election Scheme. In *Advances in Cryptology – EUROCRYPT 1997*. 103–118.

[15] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012 (LNCS)*, Vol. 7417. 643–662.

[16] Y. Desmedt and Y. Frankel. 1991. Shared generation of authenticators and signatures. In *Advances in Cryptology – CRYPTO 1991*. 457–469.

[17] Y. Desmedt and S. Jajodia. 1997. *Redistributing secret shares to new access structures and its applications*. Technical Report ISSE TR-97-01. George Mason University.

[18] Y. Deswarte, L. Blain, and J.-C. Fabre. 1991. Intrusion tolerance in distributed computing systems. In *IEEE Symposium on Security and Privacy*. 110–121.

[] S. Duan, M. K. Reiter, and H. Zhang. 2017. Secure causal atomic broadcast, revisited. In *International Conference on Dependable Systems and Networks (DSN)'17*. 61–72.

[19] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[20] P. Feldman. 1987. A practical scheme for non-interactive verifiable secret sharing. In *28th IEEE Symposium on Foundations of Computer Science (FOCS)*. 427–438.

[21] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. 1997. Optimal-resilience proactive public-key cryptosystems. In *IEEE Symposium on Foundations of Computer Science (FOCS) 1997*. 384–393.

[22] M. Franklin and M. Yung. 1992. Communication Complexity of Secure Computation (Extended Abstract). In *ACM Symposium on Theory of Computing (STOC) 1992*. 699–710.

[23] G. R. Ganger, P. K. Khosla, M. Bakkaloglu, M. W. Bigrigg, G. R. Goodson, S. Oguz, V. Pandurangan, C. A. N. Soules, J. D. Strunk, and J. J. Wylie. 2000. Survivable storage systems. *IEEE Computer* 33 (2000), 61–68. Issue 8.

[] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. 2019. SBFT: a scalable and decentralized trust infrastructure. In *49th International Conference on Dependable Systems and Networks (DSN)*.

[25] M. P. Herlihy and J. D. Tygar. 1988. How to make replicated data secure. In *Advances in Cryptology – CRYPTO '87 (LNCS)*, Vol. 293. 379–391.

[26] M. P. Herlihy and J. M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12 (July 1990), 463–492. Issue 3.

[27] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. 1997. Proactive Public Key and Signature Systems. In *4thACM Conference on Computer and Communications Security (CCS)*. 100–110.

[28] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. 1995. Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In *Advances in Cryptology – CRYPTO '95*, Vol. 963. 339–352.

[29] A. Kate, G. M. Zaverucha, and I. Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology – ASIACRYPT 2010 (LNCS)*, Vol. 6477. 177–194.

[30] E. Kokoris-Kogias, E. C. Alp, S. D. Siby, N. Gailly, L. Gasser, P. Jovanovic, E. Syta, and B. Ford. 2018. CALYPSO: Auditable Sharing of Private Data over Blockchains. Cryptology ePrint Archive, Report 2018/209. https://eprint.iacr.org/2018/209.

[31] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. 2009. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems* 27, 4 (2009).

[32] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin. 2008. Zyzzyva: speculative byzantine fault tolerance. *Commun. ACM* 51, 11 (2008), 86–95.

[33] H. Krawczyk. 1994. Secret sharing made short. In *Advances in Cryptology – CRYPTO '93 (LNCS)*, Vol. 773. 136–146.

[34] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song. 2019. CHURP: Dynamic-Committee Proactive Secret Sharing. Cryptology ePrint Archive, Report 2019/017. https://eprint.iacr.org/2019/017.

[35] J.-P. Martin and L. Alvisi. 2006. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (July 2006), 202–215.

[36] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy 2013*. 397–411.

[37] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. 2016. The honey badger of BFT protocols. In *23rd ACM Conference on Computer and Communications Security (CCS)*. 31–42.

[38] M. Naor, B. Pinkas, and O. Reingold. 1999. Distributed pseudo-random functions and KDCs. In *Advances in Cryptology – EUROCRYPT '99 (LNCS)*, Vol. 1592. 327–346.

[39] V. Nikov and S. Nikova. 2005. On Proactive Secret Sharing Schemes. In *International Workshop on Selected Areas in Cryptography 2004*. 308–325.

[40] M. Nojoumian and D. R. Stinson. 2013. On dealer-free dynamic threshold schemes. *Advances in Mathematics of Communications* 7, 1 (Feb. 2013), 39–56.

[41] P. S. Nordhold and M. Veeningen. 2018. Minimising communication in honest-majority MPC by batchwise multiplication verification. In *International Conference on Applied Cryptography and Network Security (ACNS) 2018 (LNCS)*, Vol. 10892. 321–339.

[42] T. P. Pedersen. 1992. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology – CRYPTO '91 (LNCS)*, Vol. 576. 129–140.

[] M. K. Reiter and K. P. Birman. 1994. How to securely replicate services. *ACM Transactions on Programming Languages and Systems* 16, 3 (May 1994), 986–1009.

[] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer. 2019. Scalable and Probabilistic Leaderless BFT Consensus through Metastability. *arXiv preprint arXiv:1906.08936* (2019).

[43] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE Symposium on Security and Privacy 2014*. 459–474.

[44] D. Schultz, B. Liskov, and M. Liskov. 2010. MPSS: Mobile Proactive Secret Sharing. *ACM Transactions on Information and System Security* 13, 4 (Dec. 2010), 1–32.

[45] A. Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613.

[46] M. Tompa and H. Woll. 1988. How to share a secret with cheaters. *Journal of Cryptology* 1 (1988), 133–138.

[47] T. M. Wong, C. Wang, and J. M. Wing. 2002. Verifiable Secret Redistribution for Archive Systems. In *International IEEE Security in Storage Workshop 2002*.

[] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM Conference on Principles of Distributed Computing (PODC)*.

[48] L. Zhou and F. B. Schneider. 2005. APSS: Proactive Secret Sharing in Asynchronous Systems. *ACM Transactions on Information and System Security* 8, 3 (Aug. 2005), 259–286.

A Security

In this appendix, we show why our composition of VSSR along with PBFT is secure. We do this by first observing under what conditions the share recovery protocol will terminate. To show linearizability and liveness, we map every execution of our modified PBFT algorithm to the original PBFT algorithm. Thus, since the original PBFT algorithm satisfies linearizability and liveness, so does our modified algorithm. Then, we show privacy separately.

Share Recovery Protocol Termination. We claim that the share recovery protocol will always terminate if $f + 1$ replicas have successfully completed the sharing and the network eventually delivers all messages. To see why, recall that a replica that is missing its share needs the output of vssRecoverContrib* from $f + 1$ replicas. If $f + 1$ replicas are honest, then they will faithfully call vssRecoverContrib* and send the output to a replica that is missing its share. The missing share can then be recovered by using vssRecover* to terminate the share recovery protocol.

Normal Case Protocol In the normal case protocol, we only have changed how the client constructs requests. If a client is honest, then we can simply ignore the secrets being shared in the request and have the client send regular requests in the original run of the PBFT algorithm. The requests are consistent due to the binding property of our verifiable secret sharing (VSS) scheme. If a client is dishonest and sends an invalid share to the replica, then in the original PBFT protocol run the client will drop the request message to the replica. Now, when the message is dropped, the replica can still obtain the request from another replica in the system. In the modified protocol, this is done using the share recovery protocol so we simply wait to deliver the request messages in the original run until the share recovery protocol terminates. Note that if the share recovery protocol never terminates in the modified protocol, that means that less than $f + 1$ honest replicas have the request. This means that strictly less than $2f + 1$ total replicas have the request, making it impossible for this request to be prepared. Therefore, if the modified normal case protocol never terminates, then neither does the original protocol. Thus, we see that liveness is unchanged from the original PBFT protocol.

Additionally, through the binding property of our underlying VSS scheme, we know that if a request has been committed, all secret values must be consistently shared. Thus, we see that the linearizability property also follows from linearizability in the original protocol along with binding.

Checkpoint protocol The checkpoint protocol is identical to a case where the state of the replicated service contains only the commitments of the secret values instead of the secret values themselves. Thus, by the binding property of the underlying VSS scheme, we have a one to one mapping from a run of the checkpoint protocol for our modified PBFT algorithm and the original PBFT algorithm. Therefore, if the original PBFT's checkpoint algorithm provides liveness and linearizability, then so does our modified algorithm.

State Transfer Protocol The state transfer protocol can be mapped back similarly to the normal case protocol. A replica receiving the value of a key using the share recovery protocol in our modified PBFT would have been receiving the plaintext value of the key in the original PBFT algorithm. We simply delay the plaintext value of the key until the share recovery protocol completes in our modified PBFT protocol. Additionally, in the state transfer protocol, we know that the share recovery protocol will complete since at least $2f + 1$ replicas have the state at the last checkpoint. This means that at least $f + 1$ honest replicas have the state, which is sufficient to guarantee termination during periods of synchrony.

Privacy Our modified PBFT protocol achieves privacy through the hiding property of the VSS protocol. The hiding property says that a legitimate adversary (i.e. one that has at most f shares of the secret) cannot do nonnegligibly better than guessing the secret at random. Thus, privacy is satisfied unless an adversary gets at least $f + 1$ shares of a value. However, this means that some correct replica has shared the secret with the adversary which contradicts our threshold assumption. Thus, we see that our modified PBFT protocol preserves linearizability and liveness while also guaranteeing privacy.

B Instantiation of Pedersen VSS

We describe how to fit the secret sharing scheme from Pedersen [42] into our framework.

- vssInit($1^\kappa, k, n$) first chooses a safe prime $q = 2q' + 1$ at least κ bits in length, for q' a prime. Also, we let g and h be two distinct generators of the quadratic residues $QR(\mathbb{Z}_q^*)$ of $\mathbb{Z}_q^*$ such that $\log_g(h)$ is unknown. Then, vpk_i is set to $\langle g, h \rangle$ for all i. vsk_i is set to $\perp$ for all i. The return value of vssInit is $\langle q, \{\langle vpk_i, vsk_i \rangle\}_{i \in [n]} \rangle$.
- vssShare($s, q, \{vpk_i\}_{i \in [n]}$) first extracts the public key and gets g and h as defined in vssInit. Set s_j be the coefficient of the x^j term in s and $s(i)$ be the evaluation of s at point i. Pick $t \in \mathbb{Z}_q[x]$ to be a random polynomial of degree $k - 1$. Similarly, we let t_j be the coefficient of x^j term in t and $t(i)$ be the evaluation of t at point i. Now, we set u_i to be $\langle s(i), t(i) \rangle$. Here, all linear operations on u_i values are just done element-wise. Set c to be $\{g^{s_j} h^{t_j}\}_{j \in [k]}$. Then, vssShare returns $\langle c, \{u_i\}_{i \in [n]} \rangle$.
- vssVerify(vpk_i, c, u_i) first extracts $\{g^{s_j} h^{t_j}\}_{j \in [k]}$ from c. Then, $s(i), t(i)$ is extracted from u_i. We return true if $g^{s(i)} h^{t(i)} = \prod_{j=0}^{k-1} (g^{s_j} h^{t_j})^{i^j}$ and false otherwise.
- vssReconstruct($c, \{\langle vpk_i, u_i \rangle\}_{i \in I}$) first calls vssVerify($vpk_i, c, u_i$) for all $i \in I$. If all of vssVerify calls return true, then we continue. Otherwise, vssReconstruct returns $\perp$. Then, we extract $s(i), t(i)$ from each u_i. Finally, we simply do a Lagrange interpolation in order to identify the unique degree $k - 1$ polynomial in $\mathbb{Z}_q[x]$ that goes through the points $(i, s(i))$ for all $i \in I$ and return that value.
- vssMakeSecret($q, \{\langle x_i, y_i \rangle\}_{i \in I}$) is identical Kate et. al.'s [29] instantiation described in Section 3.7.1.
- vssCombineCommitments($c, \hat{c}$) extracts $\{g^{s_j} h^{t_j}\}_{j \in [k]}$ from c and $\{g^{\hat{s}_j} h^{\hat{t}_j}\}_{j \in [k]}$ from $\hat{c}$. Then, it returns $\check{c} = \{(g^{\hat{s}_j} h^{\hat{t}_j})(g^{s_j} h^{t_j})\}_{j \in [k]}$.

C PBFT State Transfer and View Change

In this section, we describe the PBFT state transfer and view change protocols.

C.1 View Change Protocol

The view change protocol changes the leader. The core mechanism for transferring safe values across views is for a new leader to collect a set P of view-change messages from a quorum of $2f + 1$ replicas. Each replica sends a view-change message containing the replica's *local state*: Its local request-log, and the commit-certificate with the highest view number it responded to with a commit message, if any.

The leader processes the set P as follows.

(1) Initially, it sets a *leader-log* G to an empty log.
(2) If any view-change message contains a valid commit-certificate, then it selects the one with the highest view number and copies its log to G. Share recovery is triggered for any requests in G that the leader is missing its private share.

The leader sends a new-view message to all replicas. The message includes the new view number, the set P of view-change messages the leader collected as a *leader-proof* for the new view, and the *leader-log* G. A replica accepts a new-view message if it is valid and *adopts* the leader log. It may need to roll back speculatively executed requests, and process new ones. As usual, processing may entail triggering share-recovery for any requests where the replica is missing its private share.

C.2 State Transfer Protocol

We present a modified version of the PBFT state transfer protocol that is simpler and more suited when TCP is used for the underlying network protocol. When a replica has fallen behind, it sends a *state transfer request* along with its current sequence number to at least $f + 1$ replicas. Some replica will respond with the most recent valid checkpoint messages and the messages from the normal case protocol that were missed by the slow replica. In addition, the response will contain only the values of the keys that have changed since the sequence number known to the slow replica as well as the full requests that came after the last checkpoint.