Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares

Fadhil I. Kurnia
University of Massachusetts Amherst
United States
fikurnia@cs.umass.edu

Arun Venkataramani
University of Massachusetts Amherst
United States
arun@cs.umass.edu

ABSTRACT

State-of-the-art consensus protocols like Paxos reveal the values being agreed upon to all nodes, but some deployment scenarios involving a subset of nodes outsourced to public cloud providers motivate hiding the value. In this work, we present the *primary-backup secret-shared state machine* (PBSSM) architecture and an underlying consensus protocol *Oblivious Paxos* (OPaxos) that enable strong consistency, high availability, privacy, and fast common-case performance. OPaxos enables privacy-preserving consensus by allowing acceptors to safely agree on a secret-shared value without utrusted acceptors knowing the value. We also present *Fast Oblivious Paxos* (Fast-OPaxos), which enables consensus over secret-shares in three one-way delays under low concurrency settings. Our prototype-driven microbenchmarks and smarthome case study show that OPaxos induces a negligible latency overhead of at most 0.1 ms compared to Paxos while maintaining more than 85% of Paxos' capacity for small requests, and can provide lower latency and higher capacity compared to Paxos for large request sizes.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Security and privacy**;

KEYWORDS

Distributed Consensus, Privacy, Secret-Sharing

ACM Reference Format:

Fadhil I. Kurnia and Arun Venkataramani. 2023. Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624647

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624647

1 INTRODUCTION

Consensus is a fundamental building block for highly available distributed systems with strong consistency, however state-of-the-art consensus protocols like Paxos [33] and others suffer from a critical privacy drawback: the agreement protocol reveals the value being agreed upon to all replica servers. Traditional approaches such as the replicated state machine (RSM) [55] need replicas to be able to view the underlying application state in order to execute requests. But exposing the request values and state is problematic in deployment scenarios managing sensitive information and outsourcing infrastructure to untrusted cloud providers that may be *honest but curious*.

Building general consensus-based services managing sensitive information on untrusted public clouds is challenging. Purist approaches to this end include secure multiparty computation (SMPC) [15] and fully homomorphic encryption (FHE) [17] that are powerful in their generality, but despite much progress on both fronts in recent years, remain prohibitively costly for general services today. Secure computing hardware, e.g., Intel's SGX, combined with oblivious RAM techniques [18] is another option in the design space with lower overhead but stronger trust assumptions and weaker privacy guarantees. None of these offer *information-theoretic privacy* [57], i.e., resilience against an adversary with unbounded computation power, our overarching design goal, while ensuring strong consistency, high availability, and fast common-case performance for general services, a quartet of design goals that is fundamentally challenging.

Hybrid clouds, a widely used infrastructure setup today, offer a distinct opportunity to come closer to the above ideal quartet of design goals. Hybrid cloud deployments involve a subset of trusted servers on premises while relying on untrusted cloud servers for availability despite failures. The availability of trusted servers in the common case makes it easier to maintain fast common-case performance provided we can smoothly failover to untrusted servers alone while continuing to preserve as much of the applications' functionality and as close to information-theoretic privacy as is practically feasible. Our position is that this point in the design space offers better tradeoffs in practice to the alternative of not having information-theoretic privacy at all and/or

being limited to services for which the overhead of SMPC, FHE, or secure computing infrastructure is acceptable.

To that end, we present a novel architecture, *primary-backup secret-shared state machine* (PBSSM), for building highly available services with strong consistency requirements that ensures information-theoretic privacy of request values as well as underlying state in the common-case when at least one trusted server is available. In the event of failure of all trusted servers, PBSSM gracefully fails over to one of two modes: (1) a client-driven mode suitable for applications with client-partitioned state (e.g., key-value store operations with keys identifying clients) wherein a trusted client can play the role of a trusted primary; or (2) more general operations relying on SMPC with commensurate overhead and *real-world ideal-world* privacy (weaker than information-theoretic privacy as detailed in §3.1).

The key technical innovation that drives PBSSM is *Oblivious Paxos* (OPaxos), a privacy-preserving consensus protocol that provides information-theoretic privacy by integrating secret-sharing into the Paxos family of consensus protocols while preserving its traditional safety and liveness properties. OPaxos uses (t, n) threshold secret-sharing that generates n secret-shares from a single secret value in a manner that enables us to reconstruct the secret with just t shares, for a configurable $t \in [1, n]$. The protocol requires at least t acceptors as the intersection between quorums in the two Paxos phases as opposed to the traditional non-zero intersection requirement (e.g., using majority quorums for both phases). While previous works [47, 63, 70] have leveraged t-intersection of acceptor quorums for performance reasons, to the best of our knowledge, this work is the first to ensure the privacy requirement, one that introduces subtle design differences–yet important to ensure agreement safety–while also enabling unique performance optimization opportunities (detailed in §4). OPaxos can also be used outside the PBSSM context in existing distributed storage systems [7, 19, 32, 38, 40, 59] that do use secret sharing for privacy against untrusted servers but rely on an external coordination service (entailing latency overhead) to ensure consistency under failures or concurrency by offering a general-purpose consensus protocol with in-built privacy.

Given our focus on hybrid cloud scenarios, it is natural to consider the seemingly more straightforward alternative of having the trusted consensus leader simply encrypt request values when at least one trusted server is available. However, encryption carries two fundamental drawbacks: (1) it does not offer information-theoretic privacy, instead relying on assumptions on an adversary's computational resources, and in practice induces vulnerability to key theft and/or increased key management complexity to thwart them; (2) it poses an inconvenient cost-availability tradeoff as the system must rely on an external trusted key escrow

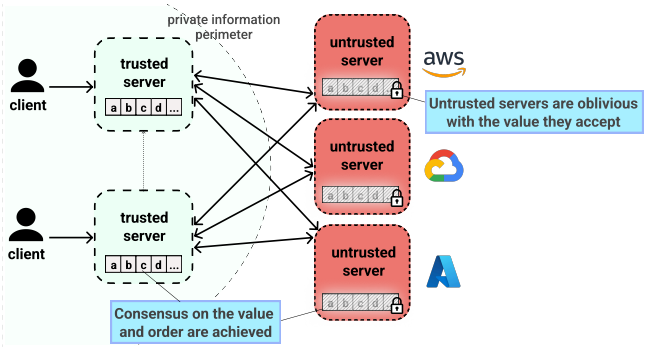


Figure 1: System and threat model: Clients' data must remain private from the untrusted servers placed at different cloud providers.

service as otherwise (permanent) failure of internal trusted nodes possessing the key can cause (permanent) data irrecoverability. Finally, as detailed in §3.1 and our experimental evaluation, secret-sharing allows us to naturally leverage recent advances in SMPC (compared to FHE) for more general services when no trusted server is available with overhead comparable to simple encryption.

We build further upon the OPaxos design to develop *Fast Oblivious Paxos* (Fast-OPaxos), an optimization similar in spirit to Fast-Paxos vis a vis classic Paxos. Fast-OPaxos enables a client to receive a commit confirmation in three one-way delays while being leader-oblivious, one less than in the case of leader-aware traditional Paxos. Fast-OPaxos design improves both latency and throughput with a single trusted proposing client or in general when conflicts are rare

We have implemented an open-source prototype for OPaxos and Fast-OPaxos, and a prototype key-value store embodying the PBSSM approach on top of OPaxos. Through prototype-driven experiments, we show that compared to Paxos, the secret-sharing overhead in OPaxos entails only a modest latency and capacity overhead, in part because of the reduced size of secret shares as well as hardware support such as SIMD and AES-NI in modern CPUs enabling fast cryptographic operations. For small request values, our OPaxos prototype induces a latency overhead of less than 0.1 ms while providing over 85% of non privacy-preserving Paxos' capacity; while for large requests, OPaxos can *improve* both latency and capacity.

We further demonstrate the end-to-end benefits of OPaxos via a case study of a modern smart-home system with trusted servers residing within the home and the traditional smart-home cloud service distributed across multiple (non-colluding) cloud providers, thereby providing high availability and linearizable consistency while limiting the perimeter of private information leakage to within the trusted home zone.

In summary, our primary contribution is a novel architecture, primary-backup secret-shared state machine, to build

fault-tolerant distributed services with strong consistency while ensuring information-theoretic privacy, and comprises the following technical sub-contributions:

(1) Design and implementation of OPaxos (§4) with a rigorous formal proof of safety as well as model checking (§4.4), and a Fast-OPaxos optimization (§4.5);
(2) Prototype-driven evaluation of OPaxos (§7) showing modest latency and capacity overhead in general and improved latency and capacity for large requests;
(3) Case study evaluation of a smart-home system based on OPaxos (§8.2) in a multi-cloud deployment setting.

2 PROBLEM MODEL AND BACKGROUND

This section describes our system, threat and failure model.

2.1 System, Threat, and Failure Models

System model. Our target scenario, as illustrated in Figure 1, is a highly available general service (or state machine) to manage private client data that is provided by a collection of n servers some of which are *trusted* and others *untrusted*. The untrusted servers must not know anything about the managed client data or client requests that act upon that state. The trusted subset of servers implement service execution, i.e., computation and state management in response to client requests, while the untrusted servers may be relied upon to ensure availability in the face of temporary or permanent server failures. The system must ensure strong linearizability consistency[1]. An example scenario is a smart-home management service wherein the trusted nodes reside within the home limiting the perimeter of sensitive information leakage to the home; another is a distributed password management or a client account management system in a hybrid cloud setup wherein the trusted nodes are on-premises while the untrusted nodes are on third-party clouds. Lack of strong consistency in such systems can compromise safety, e.g., they can compromise security in a smart-home if the surveillance devices or smart locks do not reflect the most recent configured routines.

Threat and failure model. The trusted servers and clients mutually trust each other. The untrusted servers are assumed to be honest but curious, i.e., they are expected to execute the protocol correctly but may try to glean any information they can from received protocol messages. The network environment is asynchronous and any node may experience a temporary crash or a permanent failure that may render any state on it irrecoverable. The non-byzantine crash failure model is consistent with the honest-but-curious threat model for untrusted servers, nevertheless untrusted servers may collude with each other provided less than t untrusted servers collude, where t is an a priori known limit. Our threat model precludes side-channel attacks based on message size, timing, etc. Both encryption-based and information-theoretically private approaches in general are susceptible to such side-channel attacks and require additional mechanisms extensively studied by others to thwart them [14, 41, 42] that are outside the scope of this work.

[1]This basic model also generalizes to services with weaker consistency semantics that can be satisfied using consensus as a building block.

2.2 Background Primers

OPaxos builds upon Paxos so as to offer information-theoretic privacy among other design goals. We include brief primers of Paxos [34] and information-theoretic privacy [57].

Paxos is an asynchronous distributed consensus protocol enabling a fixed set of nodes to propose values and agree upon one of those proposals as the chosen decision. Paxos proceeds in increasing, ordered rounds (also called ballots) many-to-one mapped to proposing nodes. A proposing node attempts to complete two phases in its current round: (i) a *prepare* phase wherein a *proposer* attempts to get a *prepare quorum* (or Q1) of *acceptors* to affirm its round number by promising not to accept values in lower rounds and report their respective accepted values if any in the highest lower round; and subsequently (ii) an *accept* phase in which it actually proposes its value and seeks to get an *accept quorum* (or Q2) of *acceptors* to accept that value while respecting their respective promises in the prepare phase. Paxos ensures the safety property that only a single proposal can be chosen as the decision by restricting the proposable value in the second phase based on the values if any reported in the first phase. Paxos requires that the intersection of any Q1 quorum and Q2 quorum is nonempty (e.g., both majorities).

Information-theoretic privacy guarantees that no information about protected data is revealed to adversaries even with unlimited computational resources and time. Information-theoretic privacy-preserving schemes, of which secret-sharing is a well-known example, do not rely on a key and ensure that any adversary with less than the threshold number of shares would find all potential secret values equiprobable, thereby learning no information about the secret [57]. In comparison, encryption-based approaches necessarily make limiting assumptions about the adversary's computation power, e.g., hardness of factoring the product of large primes, to ensure privacy [13]. Furthermore, in practice, encryption-based approaches are also vulnerable to key theft and/or entail additional key management complexity to thwart them, and must rely on a key escrow mechanism to protect against accidental key loss that can potentially render encrypted data permanently unavailable.

3 OPAXOS SYSTEM ARCHITECTURE

OPaxos targets the following design goals, the combination of which is both novel and nontrivial to achieve

Mode	Application class	Privacy property	Mechanism	Example scenarios
Trusted	General State Machine	Information-Theoretic	PBSSM (§3.2)	Warehouse system (§7.4), Smart-home system (§8.2)
Untrusted	Client-Partitioned State Machine	Information-Theoretic	CPSSM (§5.1)	Private key-value store (§7.5)
	General State Machine	Ideal World/ Real World	SMPC (§5.1)	Private data analytics [38, 49, 66]

Table 1: An OPaxos-based system's operational modes.

(1) **Common-case performance**: Low OPaxos overhead when at least one trusted server is up.
(2) **Seamless failover**: Seamless high availability despite server failures including failure of all trusted servers.
(3) **Information-theoretic privacy**: Ensuring that untrusted servers, even with unbounded resources, do not learn anything about the client state or requests.
(4) **Strong consistency**: Support for general applications with strong state consistency constraints.

Strictly achieving all of the above goals with approaches known today is very challenging, if at all possible, so OPaxos' goal is to come as close to them as possible. To better understand the challenge, let's consider a few natural alternatives.

3.1 State-of-the-art Alternatives Analysis

A straightforward approach is encryption wherein the trusted servers can be viewed as a trusted proxy commonly used in privacy-preserving data stores that encrypts client requests before agreeing upon their order in coordination with the untrusted backups. Encryption-based approaches however suffer from several drawbacks. First, they do not afford information-theoretic privacy by definition making them vulnerable to key compromise, theft, or trapdoors. Second, they necessitate additional infrastructure in the form of a key escrow service as otherwise permanent failure (say disaster-induced) of trusted servers or otherwise loss of the encryption key can result in permanent data unavailability. More importantly, they poorly meet the seamless failover design goal as in the absence of any trusted server, untrusted servers need to be able to make progress with encrypted copies of state. Supporting general state machine services in this mode necessitates fully homomorphic encryption (FHE), but the set of applications for which FHE techniques are low-overhead enough to be practical today is rather limited.

In comparison to FHE, advances in secure multiparty computation (SMPC) in recent years show significant speedup [15] and have rapidly expanded the class of computations that can be performed with overheads low enough to be usable in practice, e.g., systems such as SECRECY [38], Obscure [19], Senate [49], Conclave [66], and others have expanded the scope of computation from simple operations like boolean and arithmetic operations to richer functions like those in typical database query languages (such as `select`, `count`, `limit`, `join`, etc.) as well as to optimize the query processing plan for sophisticated database queries all while maintaining privacy of the underlying state from untrusted servers.

However, the privacy afforded by both SMPC- as well as FHE-based approaches have a critical fundamental limitation, namely they do not afford request privacy, only state privacy, in a state machine. In either approach, the function being computed (locally in FHE and distributed with interactive rounds in SMPC) is by design public to the untrusted servers. Furthermore, SMPC in general also exposes the result of the computation (although for specific functions, it may be possible to orchestrate the computation so that individual untrusted servers only produce secret shares of the result that can be re-assembled by a trusted end-client). Although state-of-the-art SMPC techniques internally employ secret sharing, they also do not provide information-theoretic privacy, rather they enable ideal-world/real-world privacy [15] that guarantees that untrusted servers don't learn anything more about client data compared to what they would have learned by submitting the request to a trusted server and obtaining the result of the computation from it.

In keeping with its stated design goals, the overhead limitations of general private computation techniques, and the availability of trusted servers in the common case in many real-world scenarios, OPaxos adopts a pragmatic approach that, in the common case of trusted server availability, enables information-theoretic privacy (including request privacy) with overhead comparable to traditional consensus-based fault-tolerance. When no trusted servers are available, OPaxos continues to provide information-theoretic privacy for a class of services referred to as *client-partitioned* state machines (detailed in §5.1). For general state machines under no trusted server availability, OPaxos' secret-sharing based approach enables it to seamlessly fall back on SMPC-based alternatives and their (weaker) ideal-world privacy guarantee and overhead limitations. Our position is that this design pushes the envelope closest to the OPaxos' targeted design goals. The next section explains this design.

3.2 OPaxos/PBSSM High-level Design

OPaxos adopts a *primary-backup secret-shared state machine* (PBSSM) design wherein the trusted subset of servers (possibly singular) store application state in plaintext and untrusted backup servers store corresponding secret shares. Under graceful conditions when at least one trusted server is available, a trusted server is designated as the consensus leader (or *primary*) with the following key differences from a traditional RSM design in that: 1) the primary first executes the (tentative) next client request before agreement

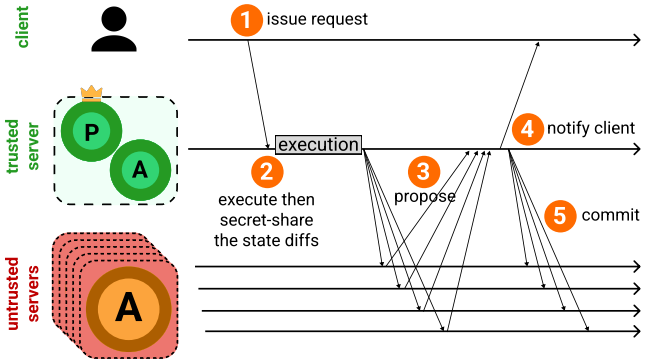


Figure 2: PBSSM execution in the default trusted mode.

over its order; and 2) the primary uses the *state diffs* or secret shares thereof resulting from the execution as its proposal for agreement over the next slot number; 3) upon agreement termination, untrusted backups apply secret shares of the agreed upon state diffs while trusted backup servers apply plaintext diffs to their copy of the state, and the primary replies back to the request-issuing client. Figure 2 illustrates this high-level PBSSM design.

A few remarks are in order for the choice of the above design, why it works, and its tradeoffs. The primary-backup design is well aligned with our threat model and assumption of common-case trusted server availability. Unlike a traditional RSM, untrusted OPaxos servers only store secret shares of application state and therefore cannot directly execute requests to compute state transformations locally, however trusted servers with plaintext state can simply execute requests and compute and transmit state diffs to the backups, so OPaxos' leader election accordingly prioritizes trusted servers, if any are available, over untrusted ones. A primary-backup approach also has the desirable dual side-effects of reducing the replication cost of execution and making the state machine deterministic as only a single trusted primary executes a request; all non-primaries, including trusted ones, apply state diffs transmitted by that primary. The justification for performing execution before agreement is more subtle and has to do with ensuring state convergence safety with non-deterministic state machines (as detailed in §5.2).

OPaxos' design enables it to seamlessly fail over from a primary-backup to a decentralized secret-shared state machine with untrusted backups alone when no trusted server is available. This mode of execution, referred to as the *untrusted* mode, for general state machines relies on secure multiparty computation, but simple applications such as key-value stores can make do with a trusted client dealer without SMPC techniques. The SMPC mode fundamentally entails the necessary limitations of requiring state machine determinism as well as lack of request privacy, neither of which limits either of the (trusted) PBSSM mode or client-dealer-driven untrusted modes. We describe the untrusted mode

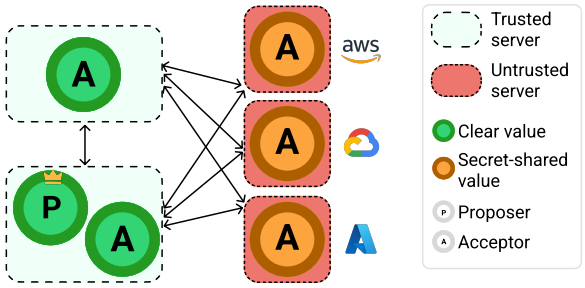


Figure 3: OPaxos' proposers and acceptors placement in the trusted and untrusted servers. Some acceptors are placed in trusted server.

operation in more detail in §5.1 and summarize the high-level operational modes in Table 1.

4 OPAXOS CONSENSUS PROTOCOL

We describe the roles, event-action protocol, quorum constraints, and formal proofs of OPaxos' consensus protocol.

4.1 Overview

As in Paxos, OPaxos has three actors: *proposers*, *acceptors*, and *learners*. A *proposer* proposes a secret-shared value to the *acceptors*, all the *acceptors* try to agree on the value based on their respective secret-shares, and the *learners* are eventually informed of the shares of the agreed upon secret value. For simplicity, we combine the *learner* and *acceptor* roles and just refer to them as *acceptor*. Since proposers perform secret-sharing, they are placed in the trusted servers allowed to know the secret value as illustrated in Figure 3. Acceptors may be placed on either trusted or untrusted servers, but untrusted acceptors must not know the secret value.

OPaxos uses (t, n) threshold secret-sharing wherein the proposer transforms a secret value into n secret-shares distributed to all the n acceptors with at least t shares required to reconstruct the secret, thereby making it resistant to $(t-1)$-collusion. A key technical challenge is to integrate threshold secret-sharing into a quorum-based consensus protocol.

Why Challenging. To appreciate the challenge, consider a strawman protocol where a proposer simply issues secret shares of the proposed value instead of the proposed value itself in a protocol otherwise identical to Paxos. This protocol has at least two problems. The first problem is the recovery of values that may have already been decided in lower round. In Paxos, if a proposer as part of the promise messages in the first phase receives even a single accepted value in a lower round, it loses the flexibility to propose an arbitrary value, however in the strawman protocol, the proposer may only retrieve a single secret share in the first phase, which is insufficient to reconstruct any lower ballot value. The second problem is Paxos safety's reliance on proposers being truthful thereby preventing them from proposing different values in

Variable Name	Description
Acceptor	
bmax	The highest ballot-number this acceptor has seen.
bacc	The ballot number in which this acceptor accepts the secret-share.
bori	The original-ballot; the first ballot number used to propose the accepted secret-share. Used as the secret value's identifier.
ssval	The accepted secret-share.
committed	A flag indicating whether ssval is committed or not
Proposer	
bcur	The current ballot number for this proposer.

Table 2: Variables in OPaxos' acceptors and proposers.

the same round, so proposing different secret shares violates that literal assumption, and furthermore, a proposer in this strawman protocol has no way to know if two secret shares received from two different acceptors were generated from the same secret value because the secret shares by design reveal no information about the original secret.

To address the first problem, OPaxos adapts the quorum constraint to ensure t acceptors intersection (not just nonzero intersection). To address the second problem of attaching additional information with a secret share that allows a node to determine if two shares came from the same secret without revealing any additional information about the secret, OPaxos relies on the *original ballot* or the ballot in which a secret share was first proposed. We explain these and other subtle adaptations including precise pseudocode next.

4.2 Consensus Protocol Phases

OPaxos, like Paxos, has two safety-critical phases initiated by a proposer in some round designated to it: *prepare* and *propose* [2]. In the prepare phase, a proposer seeks to acquire promises from some |Q1| quorum of acceptors not to accept any value in any lower round and retrieve secret shares if any that they accepted in their respective highest lower round. In the propose phase, the proposer seeks to acquire a |Q2| quorum of acceptances from acceptors for shares of the value if any retrieved from the lower round acceptances reported in the promise messages in the prepare phase, else for shares of an arbitrary value. A value is decided if a proposer acquires the designated quorums in both phases in the same round, otherwise it may retry with a higher round designated to it.

A round, also known as ballot number, is represented as a two-tuple consisting of a counter and the identity of the proposing server. Ballots define a totally ordered space wherein a ballot b_1 is greater than ballot b_2 if b_1's counter is greater than b_2's counter or if both counters are equal and b_1's identity is lexicographically "greater" than that of b_2.

Phase 1: Prepare-Promise. In the prepare phase (Phase-1) of OPaxos shown in Figure 4, a proposer sends a PREPARE(bcur)

[2]Some descriptions of Paxos refer to the *propose* phase as the *accept* phase

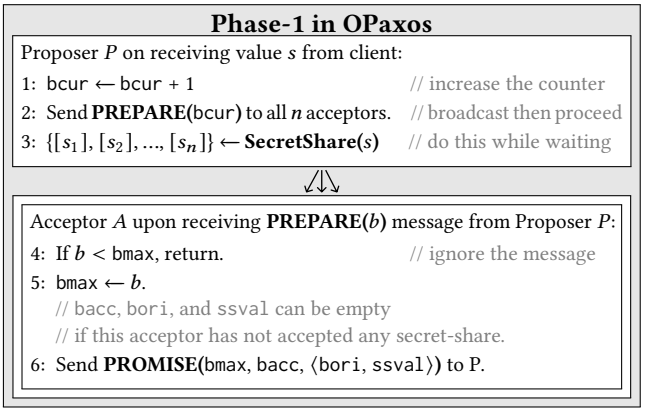


Figure 4: Phase-1: acceptors sending promises.

message to acceptors where bcur is its current ballot number. While the proposer waits for promises from acceptors, it can in parallel generate the secret-shares of value s that it wishes to propose. In all pseudocode, we use $[s_i]$ (with square brackets) to denote the i-th secret-share of s.

An acceptor maintains a total of *three* ballot numbers: (1) bmax, the highest ballot it has seen; (2) bacc, also referred to as the *accepted ballot* that is highest ballot in which it has accepted some value; (3) bori, also known as the *original ballot* that is ballot of the proposer that originally proposed the value (re-)proposed and accepted with ballot bacc. Traditional Paxos only maintains two ballot numbers at acceptors that are analogous to bmax and bacc, but OPaxos relies on bori as a connecting identifier of the secret shares of the same secret value without revealing any secret information.

Phase 2a: Recovery and Propose. The proposer waits to receive |Q1| PROMISE messages from acceptors in response to its PREPARE message. As in Paxos, if none of the PROMISEs report any secret shares accepted with a lower accepted ballot (bacc), the proposer is free to propose any value. However, if the PROMISEs did report shares accepted in lower ballots, the value recovery process is different, as shown in Figure 5, and described next.

From the |Q1| promises, the proposer needs to find a promise with the highest accepted ballot, denoted as S. The proposer then tries to find t promises, including S, whose original-ballot is the same as the original-ballot in S. If t such shares exist (line 7), the proposer needs to re-generate more shares from those t shares (line 7a) and reuse bo as the original ballot (line 7c). Else, the proposer is free to propose any value (line 8a) with the original ballot bori set to its own ballot bcur (line 8b).

Phase 2b: Accept. Upon receiving a proposal, if the proposer's ballot number is less than the highest ballot bmax the acceptor has seen so far, then the proposal is ignored (line 10), otherwise the acceptor accepts the proposal, and

Phase-2a in OPaxos

Proposer P after receiving **PROMISE(**b, ba, $\langle$bo, $[y]\rangle$**)** messages from at least |Q1| acceptors with b = bcur:

Let R be the set containing all the promises received by P.
Let S be a promise with the highest accepted ballot ba in R.
Let G be a set of at least t promises whose original-ballot bo is equal to S's original-ballot: S.bo.

// $G \leftarrow \{(\text{ba}, \langle \text{bo}, [s'_1]\rangle),$
// $(\text{ba}_1, \langle \text{bo}, [s'_2]\rangle),$
// $(\text{ba}_2, \langle \text{bo}, [s'_3]\rangle), ...\}$

7: If G exist:
  a. $\{[s'_1], [s'_2], ..., [s'_n]\} \leftarrow$ **GenMissShare(**$[s'_1], [s'_2], ..., [s'_t]$**)**
  b. SS $\leftarrow \{[s'_1], [s'_2], [s'_3], ..., [s'_n]\}$
  c. bori $\leftarrow$ S.bo
8: Else:
  a. SS $\leftarrow \{[s_1], [s_2], [s_3], ..., [s_n]\}$ // from line 3
  b. bori $\leftarrow$ bcur
9: Send proposals containing secret-shares SS and original-ballot bori. For each $[s_i]$ in SS:
  a. Send **PROPOSE(**bcur,$\langle$ bori, $[s_i]$ $\rangle$**)** to the i-th acceptor.

Figure 5: Phase-2a in OPaxos: proposer recovers and re-proposes the previously accepted secret value s' or proposes any secret value s sent by a client (in Phase-1).

Phase-2b and Commit in OPaxos

Acceptor A_i upon receiving **PROPOSE(**b, $\langle$bo, $[s_i]\rangle$**)** message from a proposer P.

10: If $b <$ bmax, return. // ignore the message
11: bmax $\leftarrow b$
12: bacc $\leftarrow b$
13: bori $\leftarrow$ bo
14: ssval $\leftarrow [s_i]$
15: Send **ACCEPT(**bacc**)** to proposer P

↘↓↙

Proposer P after receiving **ACCEPT(**b**)** messages from at least |Q2| acceptors with b = bcur:

16: Respond to the client.
17: For each $[x_i]$ in SS: // SS and bori are
  a. Send **COMMIT(**bcur,$\langle$ bori, $[s_i]$ $\rangle$**)** // from line 9 to the i-th acceptor A_i.

Figure 6: Phase-2b in OPaxos: proposer learns whether proposed secret-shares are accepted by a Q2 quorum.

updates the highest ballot bmax, the accepted ballot bacc, the original ballot bori, and the secret-share ssval in durable storage (line 11-14).

Commit Phase. If the proposer receives |Q2| ACCEPT responses from acceptors for its proposed shares, it considers the corresponding secret value as decided and issues a COMMIT message to that effect to all acceptors (line 17).

4.3 Quorum Constraints

Unlike Paxos that can use simple majority quorums or any quorum sizes |Q1| and |Q2| respectively for the two phases that ensure a nonempty intersection, OPaxos with (t, n) threshold secret-sharing requires the two quorums to intersect in at least t acceptors:

$$|\text{Q1} \cap \text{Q2}| \geq t \tag{1}$$

A conservative quorum configuration used in OPaxos to satisfy that requirement is $|\text{Q1}| = |\text{Q2}| = \lceil \frac{n+t}{2} \rceil$. For example, if we have 4 acceptors with a (2,4) threshold secret-sharing scheme, the quorum size is 3 (incidentally same as majority quorums in Paxos with 4 acceptors), a configuration that can handle one failed acceptor. Figure 7 shows several possible configurations of acceptors in OPaxos compared to Paxos.

OPaxos enables flexible quorum sizes [24] satisfying the t-intersection constraint that trade off availability for common-case performance, e.g., by making Q2 smaller and Q1 bigger, we get higher capacity and slightly lower request latency under graceful conditions, however availability is still limited by the Q1 quorum. Our implementation defaults to the ceiling/floor quorums in (2) and (3) below very slightly favoring common-case performance without hurting fault tolerance.

$$|\text{Q1}| = \left\lceil \frac{n+t}{2} \right\rceil \tag{2}$$

$$|\text{Q2}| = \left\lfloor \frac{n+t}{2} \right\rfloor \tag{3}$$

4.4 Safety, Liveness, and State Integrity

The safety and liveness properties ensured by OPaxos are identical to those of Paxos with generalized quorums (as opposed to simple majority quorums), as outlined below.

Theorem (Agreement and Validity Safety). *OPaxos ensures that at most a single value that some proposer has proposed is chosen as the decision (i.e., committed in line 17.a of Phase-2b in Figure 6).*

The formal proof of this theorem is deferred to our technical report [31]. Here we outline key differences in the proof of safety compared to Paxos. Recall that Paxos' agreement safety relies on a crucial invariant, namely, a proposer can propose a value v with ballot n iff there exists a Q1 quorum of acceptors such that either (1) no acceptor in that quorum has accepted any proposal with ballot less than n; or (2) v is the value of the proposal with the highest ballot less than n accepted by any acceptor in that quorum. Let us call a value-ballot two-tuple (v, k) as *proposable* if it satisfies either part of that disjunctive condition in the invariant.

OPaxos generalizes the definition of *proposable(v,k)* so as to preserve a similar-in-spirit invariant crucial to proving agreement safety, and depends on (in general) distinct values of original ballot and accepted ballot for a proposal as well as the secret sharing threshold:

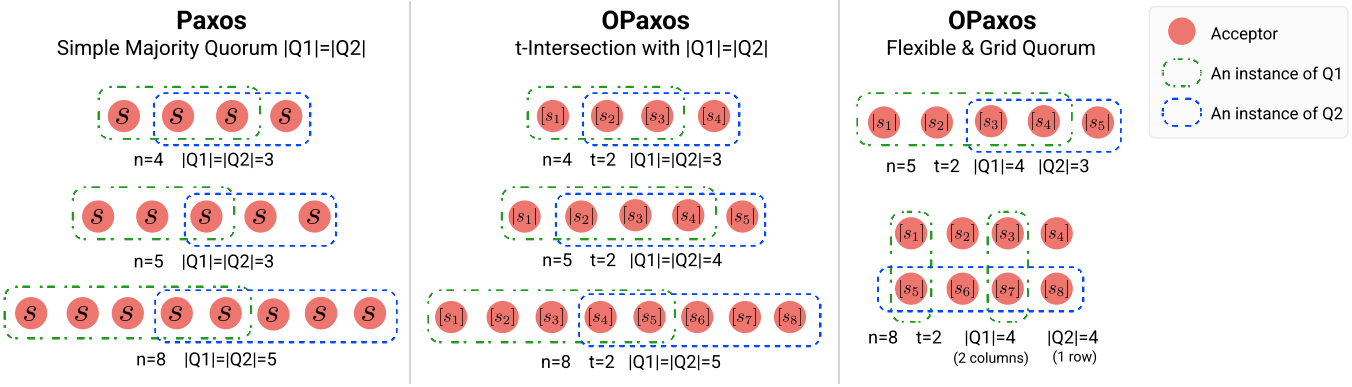


Figure 7: Example configuration of acceptor quorum set Q1 and Q2 in Paxos and OPaxos. Left: simple majority quorum, $|Q1| = |Q2| = \lceil \frac{n+1}{2} \rceil$; Middle: majority quorum with t-intersection, $|Q1| = |Q2| = \lceil \frac{n+t}{2} \rceil$; Right: flexible quorum with t-intersection, also grid quorum.

Definition (*proposable*). *proposable*(v,k) is defined as true iff there exists a Q1 quorum of acceptors such that either

i) at least t acceptors in that quorum accepted shares of v with the same original ballot in their respective highest accepted ballots less than k, and one of those t acceptors has the highest accepted ballot less than k across all acceptors in that quorum; or
ii) less than t acceptors in that quorum have accepted value shares with the same accepted ballot as the highest accepted ballot less than k across all acceptors in that quorum.

The formal proof shows that if a value v has been decided in some ballot k, *proposable*(x,j) is false for any $x \neq v$ and $j > k$, which combined with the invariant helps complete the proof. The proof relies on the t-intersection property of the Q1 and Q2 quorums in order to ensure that a decided value will always be reconstructable by a new ballot coordinator.

In our technical report [31], we also formally describe the OPaxos-driven PBSSM protocol and prove that it preserves *primary integrity*, a critical property needed to prevent state corruption or divergence despite leader changes in any primary-backup replication system [27].

Liveness. OPaxos preserves Paxos' liveness property ensuring progress when at least max(|Q1|, |Q2|) acceptors are up and can communicate in a timely manner. Termination can not be guaranteed because of the FLP impossibility result, but the protocol will terminate when a single coordinator remains uncontested and long-lived enough to complete both phases of the protocol in the same round. Furthermore, because the threshold $t \leq \min(|Q1|, |Q2|)$, a decided value will always be reconstructable during periods of liveness.

4.5 Fast Oblivious Paxos

We have additionally developed Fast Oblivious Paxos (Fast-OPaxos), which similar in spirit to Fast Paxos [36] is an optimization that reduces end-to-end delay and also places less work on the bottleneck server, thereby also making it well suited to leaderless Paxos extensions [35, 45]. A detailed description of the threshold-based quorum constraints and formal proofs of safety and liveness are available in our technical report [31].

5 OPERATIONAL ODDS AND ENDS

5.1 Untrusted Mode Operation

The description thus far assumed that at least one trusted server is available to act as the PBSSM primary. We next describe the operation of an OPaxos-based system when no trusted server is available, and further separate it into two application sub-categories: 1) general state machines; 2) client-partitioned state machines, explained in turn below. The former can support arbitrary services but entails more overhead while the latter is more suitable for services whose underlying state is logically partitioned into small units mapped to corresponding end-clients that manage that state, for example, a key-value store wherein each key (or bag of keys) is mapped to a client that owns and manages the key-value pair(s). In both untrusted mode categories, trusted clients are assumed to perform secret-sharing.

General. Supporting general state machine services in OPaxos' untrusted mode requires secure multiparty computation. Unlike the PBSSM mode wherein a trusted server plays three roles: *primary*, *consensus leader*, and *secret dealer*, in the untrusted mode, a client plays the role of the secret dealer and an untrusted server acts as the consensus leader, and there is no primary as untrusted servers operate as a decentralized secret-shared state machine. Being based on SMPC, this mode fundamentally cannot support *request privacy*, i.e., untrusted nodes can and need to see the contents of client requests in plaintext in order to execute the request in a distributed manner, however the underlying application state

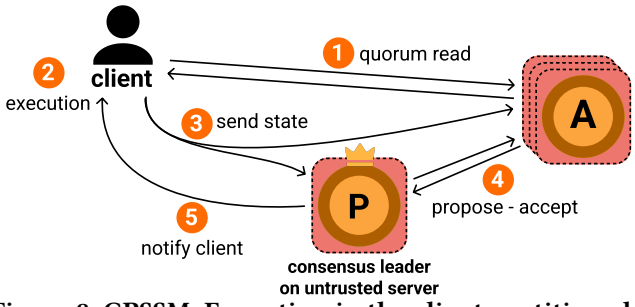


Figure 8: CPSSM: Execution in the client-partitioned secret-shared state machine.

is still secret-shared and therefore remains private from the untrusted servers as per the ideal-world/real-world model that in general is weaker than information-theoretic privacy. Specifically, SMPC in general only guarantees that untrusted servers learn no more information than they would by observing the input requests and the result of executing the requests, whereas information-theoretic privacy in OPaxos' PBSSM mode guarantees that untrusted servers can obtain no information by observing secret-shared requests as well as underlying application state.

Client-Partitioned. Supporting client-partitioned state machine services is much simpler, lower overhead, and enables information-theoretic privacy even in the untrusted mode as follows. To "execute" a request, a client dealer fetches the state partition it manages, e.g., a key-value pair, locally computes the result and re-distributes secret shares of any state modifications back to the untrusted servers (shown in Figure 8). It is straightforward to ensure request privacy by performing both read and write operations in a manner so as to appear identical, e.g., by always updating a nonce in the key-value pair even for reads to make them indistinguishable from writes. Furthermore, it is straightforward to hide the access pattern by using Oblivious RAM [18] techniques at the application level with a commensurate overhead cost.

5.2 Why Execution Before Agreement

In the trusted PBSSM mode, conducting execution before agreement or the other way round does not make a significant difference to overall request latency as either option allows the primary (a backup) to complete request handling in two (three) one-way network delays, however OPaxos' design choice prevents the possibility of state divergence when a trusted primary fails and another trusted server takes over as the consensus leader and request executing primary. The traditional sequence of agreement followed by execution decouples an already decided request from its underlying state transformation, which allows a new primary to issue state diffs different from those already applied by one or more backups and issued by the previous primary, so preventing state divergence because of nondeterministic request execution would require backups to support an *undo* operation to roll back and re-apply state diffs and for the protocol to guarantee detection of such potential divergence.

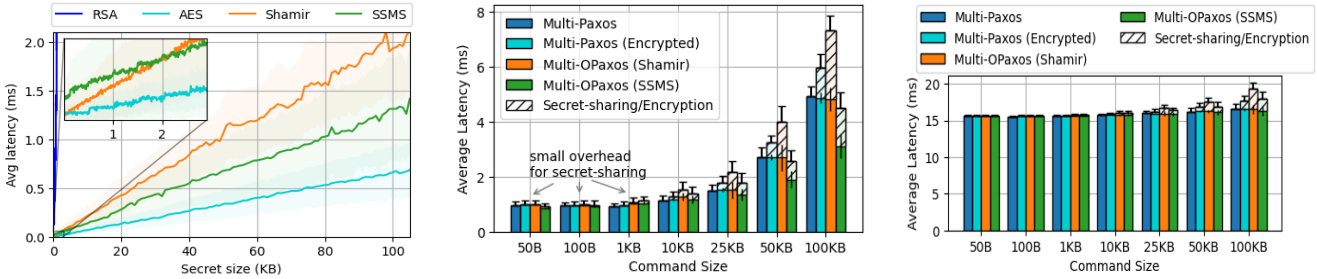
In contrast, with execution before agreement, undos are unnecessary for correctness at backups. A primary that crashes and recovers can simply roll forward from the most recent checkpoint only up until the highest cumulatively agreed upon slot number and follow the new primary's lead on subsequent state diffs. In the rare event of a new primary taking over after mistakenly suspecting the old primary as having failed (say because of asynchrony), a rollback at the old primary may still be necessary in which case OPaxos simply rolls forward the old primary from the most recent checkpoint (effectively emulating a crash), however the likelihood of such false positives can be engineered to be low with long-lived primaries and does not require additional protocol mechanisms to guarantee state convergence safety.

In the untrusted mode, a corner case with client-driven execution is when only a subset of servers have received secret shares of the updated state and the orchestrating client dealer fails midway. Fortunately, as with the PBSSM mode, performing execution before agreement can alleviate this scenario as follows: for each request, a client first collects a threshold number of secret shares from a Q1 quorum of backups for its partition, verifies that they have the same version number, locally executes the request, and then proposes secret shares of the corresponding updated partition as the next proposal whose order needs to be agreed upon by the untrusted servers. Successful agreement automatically ensures availability of a threshold number of secret shares of the updated partition whenever a majority is available (as a majority outnumbers the OPaxos' threshold by design).

6 IMPLEMENTATION CONSIDERATIONS

In this section, we describe several implementation considerations applicable to OPaxos as well as Fast-OPaxos.

Tracking state diffs in trusted mode. Our prototype implementation of PBSSM uses Linux's `strace` [39] to capture state diffs while executing each command, enabling us to support general state machines with modest overhead. For each executed command, PBSSM persistently modifies the state in the file system using file IO system calls (e.g `write` and `pwrite`). Thus, we can capture any file changes, secret-share them, and broadcast them to the backups before sending the execution results to the user. Simpler applications like key-value stores can directly implement OPaxos secret-sharing at the application layer without the overhead of `strace`.



(a) Secret-sharing and encryption latency (b) Latency on LAN (c) Latency on WAN

Figure 9: a) Microbenchmark for secret-sharing and encryption latency; b) Latency overhead of secret-sharing is negligible for small secret values, for larger values SSMS reduces the latency. c) The wide-area latency dominates, making the overhead of secret-sharing small.

Preventing duplicate shares. Because proposers independently reconstruct the Shamir polynomial and all n shares using just t shares, it is possible for two different acceptors to end up receiving duplicate shares unless all trusted proposers have an a priori agreed upon deterministic way to permute shares across acceptors. Simplistic schemes like assigning the first (lowest x) share to the first acceptor or other simple mapping functions can reveal some information about the underlying polynomial, so the best practice is for trusted proposers to use an a priori known function to generate a random permutation of shares to assign to acceptors.

Faster randomization source. We learned from our implementation and prototype evaluation experience that the randomization used to generate secret-shares of a value can induce a significant overhead, which is consistent with prior work [58] in other secure systems contexts. In OPaxos, we use multiple workers for the secret-sharing process in different CPU cores, but in Linux, all those workers share the same random generator `/dev/urandom` which becomes the bottleneck. After observing this issue, we reverted to a cryptographically secure pseudorandom number generator (CSPRNG) based on the AES counter mode for which modern CPUs widely provide hardware support, e.g., AES-NI for cryptographic computation, that greatly increases the throughput of secret generation. We extended the open-source Shamir implementation of Hashicorp Vault [22] by using CSPRNG as the random source, leveraging SIMD for Galois arithmetic operation, and with more cache-friendly polynomial generation. Our open-source implementation [29], therefore does secret-sharing faster than the original.

Smaller secret-share size. OPaxos works with any threshold-secret sharing schemes, including SSMS [28] that enables secret-shares' size reduction. SSMS uses erasure-coding that reduces the shares' size by $1/t$ without revealing the value, even in a partial form. We use an open-source reed-solomon library [52] written in Go for implementing SSMS. This SSMS optimization is critical to improve the latency and capacity performance for large client request sizes.

Other optimizations. We note a few other optimizations though we have not implemented these. First, we can use grid quorums in OPaxos where the first quorum Q1 is strictly in the form of columns and the second quorum Q2 is in the form of rows, thereby further reducing the size of each quorum but at the cost of availability, as exemplified in Figure 7 (right), where with a total of 8 acceptors and $t = 2$, we only need 4 acceptors in both quorums; two columns for Q1 and one row for Q2. However, if say acceptors that store $[x_3]$ and $[x_5]$ fail simultaneously, the protocol cannot make progress as we cannot have a complete row for the first phase. Second, in some runs, we can have a smaller Q1, for example, when all the incoming promises are empty. Specifically, in Phase-1, when the proposer gets $(|Q1| - t + 1)$ empty promises, the proposer can directly start Phase-2. When that happens, waiting for the remaining $(t-1)$ promises will not change the fact that there is no recoverable value. This optimization is similar to that in PANDO [63] and can further be generalized in OPaxos, as described in our technical report.

7 EVALUATION

Our prototype for OPaxos is implemented on top of Paxi [2], an open-source research framework for evaluating consensus protocols. We wrote the protocols using Go in around 9300 LoC, and it can be accessed at [30].

In this section, we evaluate the overhead of providing the privacy-preserving property in OPaxos (§7.1, §7.2), compared to Paxos; show how our OPaxos implementation handles node failures (§7.3); and demonstrate the latency for an emulated PBSSM (§7.4). Finally, we emulate a simple key-value store deployed across different cloud providers with no trusted server (§7.5).

The implemented prototype of Paxos and OPaxos do the typical consensus over a log of requests or state diffs (multi-decree). We call these multi-decree consensus as Multi-OPaxos and Multi-Paxos, but in this report we also refer them as

OPaxos and Paxos for brevity. We use two secret-sharing schemes for OPaxos: Shamir and SSMS, and compare them with ordinary Paxos also encrypted Paxos where the value is encrypted before being proposed.

Default evaluation setup. We evaluate OPaxos with five `m510` machines in the CloudLab cluster [10]. Each node has 2.0 Ghz Intel Xeon CPUs, 64GB RAM, and is interconnected via 1 Gbps network links. The clients run on a separate machine, and by default use 50 bytes values. TCP is used for client-leader communication as well as among the consensus instances. We use two trusted servers, $t = 2$, and the majority with t-intersection to determine the quorum sizes: for $n = 5$ and $t = 2$ the quorum sizes are $|Q1| = 4$ and $|Q2| = 3$. Beside this default setup, in some experiments, we vary the value sizes, emulate state-machine computation with a standard workload, and emulate WAN latency using Linux's `tc`.

7.1 Latency Overhead in OPaxos

We measured the latency overhead under low load, i.e., with a single outstanding request at any time. The *latency* is measured from the client side as the time since the client sent the request until it receives the response. For each protocol, a single client sends 100K requests to the leader. We plot the average latency with error bars showing standard deviation.

Secret-sharing and encryption microbenchmark. We first measure the latency of threshold secret-sharing schemes we use (Shamir [56] & SSMS [28]), and also the latency of standard encryption of RSA & AES. Using one of the machine as described in the evaluation setup, we measured the latency for (5,2) threshold secret-sharing: the time needed for generating $n = 5$ shares with $t = 2$, and the latency of the RSA & AES library in Go. As shown in Figure 9a, symmetric encryption such as AES has the lowest latency, followed by Shamir and SSMS. The latency of asymmetric encryption as RSA is too high for consensus usage, so we only consider AES for encrypted Paxos. We can see that with up to 44KB secret value, the secret-sharing and encryption latency is still below 1ms. For small secret values, the secret-sharing latency for Shamir is lower than SSMS. This trend will come in handy when we later measure OPaxos' latency overhead.

End-to-end latency overhead. With a stable OPaxos leader, every received value from the clients needs to be secret-shared, which induces additional latency compared to Paxos. However, as shown in the secret-sharing microbenchmark, for small secret value we expect the latency overhead to be small. The result is shown in Figure 9b wherein for small values up to 1KB, the secret-sharing latency overhead is less than 0.1ms. In real multi-cloud deployment, that latency overhead is negligible as the inter-node latency will typically be much higher, as we also show next.

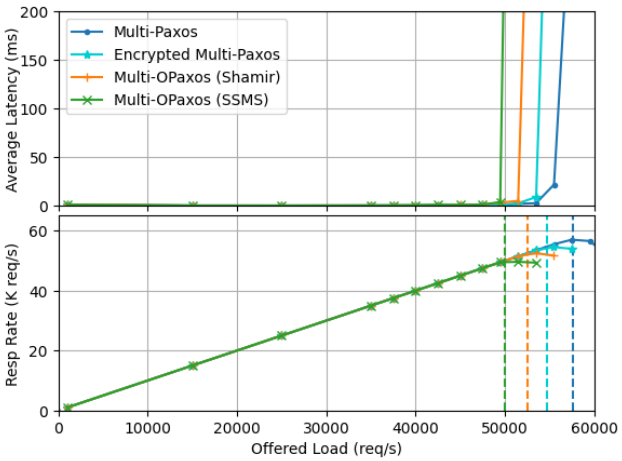


Figure 10: Secret-sharing overhead reduces the capacity of OPaxos compared to Paxos, as measured with 50 bytes values.

Consistent with the microbenchmark in Figure 9a, we also see that with larger values the latency overhead in OPaxos also increases. For OPaxos with SSMS, the benefit of secret-shares' size reduction can be seen in the measurement with large values (10-100KB). With those large values, using Shamir is slower than SSMS; this is also consistent with the result in Figure 9a. The smaller overhead of using SSMS with large values is the result of both faster secret-sharing and less data transmission from the leader to acceptors. With 50-100KB values we can even see that OPaxos with SSMS offers lower or similar latency than the non-encrypted Paxos.

Latency on wide area network. The smaller overhead of using secret-sharing in OPaxos becomes more apparent when we run the measurement in a WAN measurement. We emulate WAN deployment by having 5ms RTT in the client-leader link and 10ms RTT in the inter-node networks. As shown in Figure 9c, the wide area latency dominates, making the overhead of secret-sharing negligible.

In our technical report [31], we also analyze the impact of quorum sizes and show reduction in average latency with smaller Q2 quorum sizes (and thus larger Q1 quorums).

7.2 Capacity Overhead

In this evaluation, we measure the capacity overhead of OPaxos compared to Paxos. The secret-sharing process adds more work for OPaxos, thus OPaxos is expected to provide lower capacity than Paxos. We estimate the *capacity* of the system as the load when the difference between response-rate and the load grows more than 1%, wherein *load* is the rate of 50 bytes requests sent by the clients to the leader, and *response-rate* is the rate of responses received by the clients. We use 10 parallel clients to overload the system. We vary the load in Paxos and OPaxos, and for each load the clients

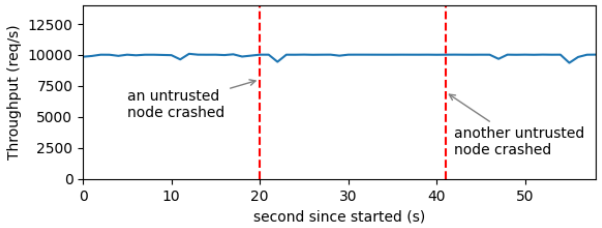


Figure 11: Failures of some untrusted servers.

send the requests for 30 seconds with inter-request times that are Poisson- distributed. We additionally also record the average latency for every load.

Figure 10 shows the load-latency measurement with the default evaluation setup. When the load is near the system capacity, we can see the response rate flattening and the latency going up drastically. In our measurement, Paxos' capacity is around 57.7K req/s; encrypted Paxos capacity is around 54.7K req/s; while the capacity of OPaxos using Shamir and SSMS are 52.5K req/s and 51.0K req/s respectively; making OPaxos' capacity with Shamir and SSMS to be 91.3% and 86.9% of Paxos' capacity. These results are close to our expectation. Measured with the same setup in a single machine, we observed the secret-sharing throughput of Shamir and SSMS are around 683.5 K req/s and 536.3 K req/s, respectively; thus, we expect our implemented OPaxos to offer more than 85% of Paxos' capacity. We also show when OPaxos with SSMS can offer higher capacity compared to Paxos in our technical report [31].

7.3 Performance Under Server Failures

Next, we show that our implementation of OPaxos is able to handle some node failures while making progress. However, compared to Paxos, the t-intersection requirement in OPaxos reduces the availability. Using the default setup with $n = 5$, $t = 2$, $|Q1| = 4$, and $|Q2| = 3$, we emulated node failures by dropping all incoming messages to some nodes.

We make the client continuously send requests to the leader with a constant load of 10,000 reqs/s, while we capture the throughput for every 100ms from the client. As shown in the Figure 11, with $n = 5$ nodes and $t = 2$, OPaxos can handle two failed untrusted nodes under a stable leader. However, the remaining three untrusted nodes after the 40s mark are not enough for leader election (Phase-1) as $f = 1$.

7.4 PBSSM Primary-Backup Computation

We try to run an arbitrary application in a primary-backup fashion using OPaxos and Paxos. We employ a primary that execute a TPC-C workload [61]. For each transaction the primary executes the queries on an sqlite database while capturing the state diffs. Then, the primary run consensus over the state diffs. With OPaxos, the primary secret-share the state diffs while in the encrypted Paxos, the primary

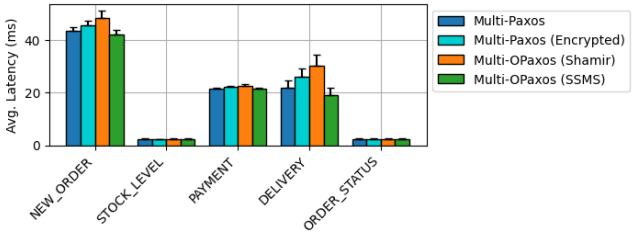


Figure 12: OPaxos and Paxos under TPCC Workload

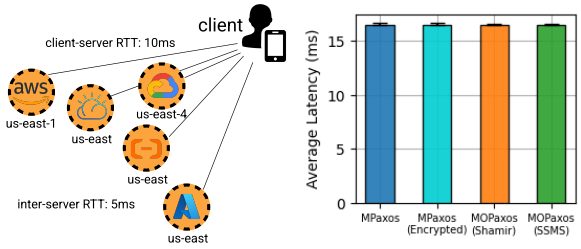


Figure 13: Key-Value Store without trusted leader. Left: emulated WAN setup. Right: client-perceived latency.

encrypts the state diffs. The result is shown in Figure 12. The latency depends on the transaction type, in general we can see that OPaxos with Shamir imposes latency overhead while OPaxos with SSMS can offers lower latency. This result is consistent with our previous latency measurements.

7.5 Key-Value Store On Untrusted Mode

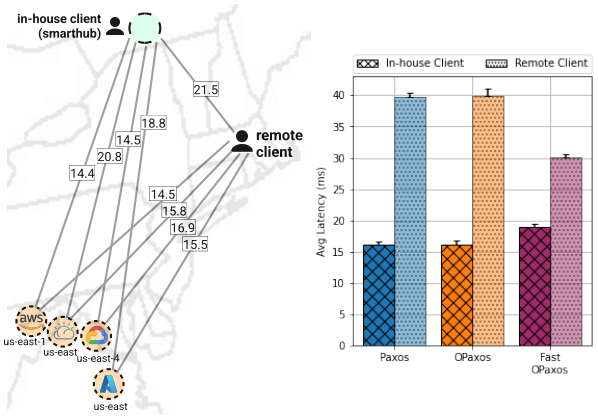
We also run a simple key-value store (KVS) with no trusted nodes available. The KVS is deployed in an emulated multi cloud providers as shown in the left of Figure 13. The client acts as secret-sharing dealer and directly broadcast the secret shares to all the nodes. One of the untrusted node acts as a leader to assign slot number to the client's proposed secret value. In the encrypted Paxos, the client only needs to send the encrypted value to the leader, not to all the nodes. As in previous measurement, we record the perceived latency from the client side under low load. The result in Figure 13 shows that the latency overhead of OPaxos is negligible.

8 SMART-HOME SYSTEM CASE STUDY

In this section, we describe a case study evaluation of OPaxos' usability in smart-home systems.

8.1 Current Smart-home System Issues

The prevailing architecture for smart-home systems today suffers from two key drawbacks: lack of privacy and lack of strong consistency guarantees, addressing which forms a key motivation for OPaxos. Privacy is a concerning issue because smart-home systems store sensitive personal data on third-party clouds making them vulnerable to breaches, as has also been observed by many prior works [4, 44, 53, 60]. To



(a) Smart-home system setup (b) Smart-home latency

Figure 14: Smart-home system setup and the ping latencies, also client perceived latencies using smart-home actions dataset.

address this privacy concern, local-first smart-home systems are being developed, for example Home-Assistant [23] and Hubitat [26] that prioritize local execution in the hub, even for features that traditionally rely on the cloud such as voice recognition [54]. However, they trade fault-tolerance for privacy by storing all the data locally in the hub making the data irrecoverable if the local storage is damaged.

Strong consistency is important because smart-home systems physically interact with users and physical systems thereby impacting their safety. For example, when there are concurrent updates to lock/unlock a smart-lock, or concurrent updates to routines managing surveillance devices, all of the backups must reflect the most recent configuration update and device state as any inconsistency could compromise physical safety. Melissaris et al. [43] shows this consistency issue in popular hubs like SmartThings and Vera where the events to unlock/lock a door can be *reordered* at multiple locations: the wireless protocol, the multi-threaded smart-home application, and in the storage layer, resulting in the door being erroneously left in a state that the user does not expect. The problem becomes more concerning when we have multiple backups that need to be synchronized. Saeida et al. [3] also highlights the importance of fault-tolerant event ordering in smart-home systems for safety.

We envision an OPaxos-based smart-home system wherein trusted home hubs/servers act as proposers and untrusted server across cloud providers act as backups, thereby ensuring strong consistency, high availability, and privacy.

8.2 Real World Workload Measurement

As a case study, we measure how OPaxos and Fast-OPaxos might perform, in terms of latency, in a real multi-cloud deployment under real smart-home systems access pattern. We instantiated virtual machine on AWS, GCP, Azure, and IBM Cloud, then measured the ping latency. We use the ping measurement results to simulate the system in our prototype by injecting delay when some nodes send messages to others; see Figure 14a for the smart-home system setup. The system has two types of clients: in-house and remote client. The in-house client resides in the house near the trusted node: the local smarthub, and the remote client is far from the house. The untrusted nodes that act as acceptors, for backup purposes, reside in different cloud providers.

We used the Mon(IoT)r [53] and PingPong [62] smart-home datasets that contain timestamped packet-captured files corresponding to actions the client performs on smart-home devices. We convert the datasets into tracefiles of read/write commands for our implemented KV store, e.g., an action to turn on a smart-device translates to a write command with value `{"state":"on"}` and device ID as key.

We got comparable latency measurement results for both datasets, which is expected since both datasets contain low-load, small-request actions typical for a smart-home system; it is rare to have concurrent clients in a smart-home system. As can be seen in Figure 14b, the overhead of doing secret-sharing in OPaxos is negligible compared to higher WAN latency. The in-house client's perceived latency for Paxos and OPaxos are 16.1ms and 16.2ms respectively. The latency for the in-house client using Fast-OPaxos is higher since a larger quorum is needed, and there is not much latency savings because the in-house client is near the leader. However, for the remote client, we can see that Fast-OPaxos offers lower latency compared to Paxos and OPaxos: the remote client experiences around 24% lower latency compared to either.

As a proof-of-concept, we have also integrated OPaxos into a popular smart-home system, Home-Assistant (HA), that supports fully-local operation. Internally, HA uses `sqlite` to store all updates. Users can manually backup the event updates [65, 67], but there might be critical updates that are missing after the last backup. There is a cloud-based backup plugin available [6], but it only backs up on a single provider and requires third-party account management. Our OPaxos-HA integration implements a simple listener for HA events provides consistent distributed backup storage while confining the perimeter of sensitive information leakage to the home, a novel combination of features that to our knowledge does not exist in any smart-home system. The purpose of this minimal proof-of-concept was to validate feasibility of integration; a more complete integration would require also implementing checkpoint/recovery mechanisms so as to maintain consistency between device state and database state that we have not implemented (but we do not foresee problems implementing them).

9 RELATED WORK

To our knowledge, this work is the first to develop a provably safe crash-fault-tolerant consensus protocol that operates

	Paxos[33]	Fast-Paxos[36]	RS-Paxos[47]	PANDO[63]	OPaxos	Fast-OPaxos
Privacy-preserving (confidentiality)	No	No	No	No	Yes	Yes
Min. intersection between Q1 and Q2	1	1	t	t	t	t
Client's Latency (message delay)	4	3 or 2*	4	6 or 4**	4	3 or 2*
Num. messages to/from the leader* **	$3n-1$	$2n$	$3n-1$	write: $2n+2$, read: $2n$	$3n-1$	$2n$
Value identifier	the value itself	the value itself	unspecified	rand. number/ hash	original-ballot	original-ballot

* Fast-Paxos and Fast-OPaxos reach consensus in one round-trip for a non-executable update when *all acceptors* directly notify the client, not the coordinator.
** PANDO does two-rounds write with delegation and one-round read under no conflict; one round-trip more is needed for client to contact *a frontend.*
*** The leader is an elected proposer in Paxos, RS-Paxos, and OPaxos; it is the coordinator in Fast-Paxos and Fast-OPaxos; it is a frontend in PANDO.

Table 3: Paxos variants as well as other consensus protocols with $t > 1$ intersection between the two phases.

over secret shares, however it builds upon a significant body of closely related work on consensus as well as privacy.

Paxos variants employing similar techniques. Using quorums with $t > 1$ intersection is not new, for example, RS-Paxos [47] and PANDO [63] rely on $t > 1$ intersection. However, their main objective is storage or latency reduction, which is achieved using erasure-coding. Likewise, CRaft [70] integrates erasure-coding into Raft [48], and adaptively replicates values either with full replication or erasure-coding depending on the number of available replicas, which enables it to achieve better availability. In contrast to these, our work primarily addresses privacy issues using secret-sharing, a goal not addressed by those works and one that introduces subtle yet important differences in the protocol design as well as implementation. Furthermore, we also develop Fast-OPaxos, an optimization to reduce the end-to-end delay of consensus to at most three one-way delays. Table 3 provides the context to position our work as compared to closely related prior work.

Secret-Sharing Systems. In the industry, many secret management systems, like Hashicorp Vault [21] and Mozilla SOPS [46], use secret-sharing to secure the master key held by multiple users. Proposed research systems, like Sieve [68], DepSKY [7], Ghostor [25] also use secret-sharing for similar purpose. However, they typically assume no concurrency or use expensive locking to prevent the inconsistency. Our proposed protocols enable concurrent proposers to broadcast secret-shares of different values while providing consistency through the agreement safety property.

Several recent works [5, 8, 64] have studied how to combine secret-sharing into Byzantine Fault Tolerant (BFT) consensus. These works address a harder technical problem than the one addressed herein by providing stronger guarantees compared to our work that targets a (benign) crash-prone failure model. However, like any BFT protocol, they induce a high quadratic message complexity, and also require a higher replication factor. They necessitate a more complex secret-sharing scheme so as to ensure the verifiability property, which requires quadratic message exchange or an external PKI. Our position is that, given industry norms and regulatory constraints, an honest-but-curious threat model resistant to $t-1$-collusion as in OPaxos is a point in the design space that offers a more practically desirable tradeoff between security guarantees and resource cost and complexity.

Other Privacy-Preserving Systems. Many systems protect user privacy in different ways, including in smart home system settings, such as: hiding the access pattern [9, 12, 20, 69], working only on encrypted data [16, 37, 50, 51], or doing private operation using secure multi-party computation (SMPC) [1, 11, 19, 38]. These approaches are complementary to OPaxos or too heavy handed for the scenarios of our interest. SMPC is a powerful approach enabling a state machine to be distributed across untrusted nodes but incurs high overhead; for instance, a recent work shows the need of 6-24 additional communication rounds for simple equality, inequality, and boolean addition operation [38]. Computing directly on encrypted data, in addition to the drawbacks of encryption-based approaches previously noted, either incurs high overhead or limits application generality.

10 CONCLUSION

We present OPaxos and Fast-OPaxos, a novel family of fault-tolerant privacy-preserving consensus protocols that allow a set of trusted and untrusted nodes to agree on secret-shared values while keeping the agreed upon values information theoretically hidden from the untrusted nodes, a system setup that is valuable for hybrid cloud deployments with honest-but-curious cloud nodes. We include rigorous formal proofs of correctness and TLA+ model checking of OPaxos. Our prototype-driven microbenchmarks show that OPaxos induces only a modest latency and capacity overhead in general, and for large requests, can even improve latency and capacity compared to traditional Paxos. Our case study evaluation shows that our approach can improve privacy in smart home systems from third-party cloud providers while maintaining high availability and strong consistency.

Our open-source implementation of OPaxos and Fast-OPaxos can be accessed at: https://opaxos.github.io.

ACKNOWLEDGEMENT

We thank the anonymous SoCC reviewers and our shepherd, Jelle Hellings, for their helpful comments and feedback. This work was supported by the National Science Foundation under grant numbers 1719386, 1717132, and 1900866.

REFERENCES

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. 2020. Blinder–Scalable, Robust Anonymous Committed Broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1233–1252.

[2] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data*. 1696–1710.

[3] Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B Terry, and Riza O Suminto. 2017. Rivulet: a fault-tolerant platform for smart-home applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 41–54.

[4] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. 2018. If This Then What? Controlling Flows in IoT Apps. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1102–1119.

[5] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 2387–2402.

[6] Stephen Beechen. 2022. Home Assistant Google Drive Backup. https://habackup.io/

[7] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Trans. Storage* 9, 4, Article 12 (nov 2013).

[8] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 163–176.

[9] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. 2022. PFirewall: Semantics-Aware Customizable Data Flow Control for Smart Home Privacy Protection. In *The Network and Distributed System Security Symposium (NDSS) 2021*.

[10] CloudLab. 2023. https://www.cloudlab.us/.

[11] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 259–282.

[12] Trisha Datta, Noah Apthorpe, and Nick Feamster. 2018. A Developer-Friendly Library for Smart Home IoT Privacy-Preserving Traffic Obfuscation. In *Proceedings of the 2018 Workshop on IoT Security and Privacy* (Budapest, Hungary) *(IoT S&P '18)*. Association for Computing Machinery, New York, NY, USA, 43–48.

[13] Whitfield Diffie and Martin E Hellman. 1976. New Directions in Cryptography. *IEEE Transactions On Information Theory* 22, 6 (1976).

[14] Hassan Eldib, Chao Wang, and Patrick Schaumont. 2014. Formal verification of software countermeasures against side-channel attacks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–24.

[15] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246.

[16] Luca Ferretti, Michele Colajanni, and Mirco Marchetti. 2012. Supporting Security and Consistency for Cloud Database. In *Proceedings of the 4th International Conference on Cyberspace Safety and Security* (Melbourne, Australia) *(CSS'12)*. Springer-Verlag, Berlin, Heidelberg, 179–193.

[17] Craig Gentry. 2010. Computing arbitrary functions of encrypted data. *Commun. ACM* 53, 3 (2010), 97–105.

[18] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 182–194.

[19] Peeyush Gupta, Yin Li, Sharad Mehrotra, Nisha Panwar, Shantanu Sharma, and Sumaya Almanee. 2019. Obscure: Information-Theoretic Oblivious and Verifiable Aggregation Queries. *Proc. VLDB Endow.* 12, 9 (May 2019), 1030–1043.

[20] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 91–107.

[21] Hashicorp. 2023. Vault by HashiCorp. https://vaultproject.io/.

[22] Hashicorp. 2023. Vault's Shamir Implementation. https://github.com/hashicorp/vault/tree/main/shamir.

[23] Home Assistant. 2023. https://home-assistant.io/.

[24] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems*.

[25] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. 2020. Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 851–877.

[26] Hubitat. 2023. https://hubitat.com/.

[27] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.

[28] Hugo Krawczyk. 1993. Secret sharing made short. In *Annual international cryptology conference*. Springer, 136–146.

[29] Fadhil I. Kurnia. 2022. Go-Shamir: Fast Shamir Secret Sharing in Pure Go. https://github.com/fadhilkurnia/shamir.

[30] Fadhil I. Kurnia. 2023. Oblivious Paxos and Fast Oblivious Paxos Prototype. https://opaxos.github.io/.

[31] Fadhil I. Kurnia and Arun Venkataramani. 2023. *Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares (Extended Version)*. University of Massachusetts Technical Report UM-CS-2023-001. Amherst, MA, USA.

[32] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. 2003. Responsive Security for Stored Data. *IEEE Transactions on Parallel and Distributed Systems* 14, 9 (2003), 818–828.

[33] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[34] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[35] Leslie Lamport. 2005. Generalized Consensus and Paxos. (2005).

[36] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[37] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA.

[38] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation*.

[39] Linux strace. 2023. https://linux.die.net/man/1/strace.

[40] Thomas Loruenser, Andreas Happe, and Daniel Slamanig. 2015. ARCHISTAR: towards secure and robust cloud based data sharing.

In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 371–378.

[41] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.

[42] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH computer architecture news* 40, 3 (2012), 118–129.

[43] Themis Melissaris, Kelly Shaw, and Margaret Martonosi. 2019. OKAPI: in support of application correctness in smart home environments. In *2019 fourth international conference on fog and mobile edge computing (FMEC)*. IEEE, 173–180.

[44] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 131–147.

[45] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372.

[46] Mozilla. 2023. SOPS: Secrets OPerationS. https://github.com/mozilla/sops.

[47] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. 2014. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 61–72.

[48] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319.

[49] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics.. In *USENIX Security Symposium*. 2129–2146.

[50] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 85–100.

[51] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. 2014. Building Web Applications on Top of Encrypted Data Using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 157–172.

[52] Klaus Post. 2023. Reed-Solomon. https://github.com/klauspost/reedsolomon.

[53] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. 2019. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 267–279.

[54] Rhasspy. 2023. Rhasspy Voice Assistant. http://rhasspy.org/.

[55] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[56] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[57] Claude E Shannon. 1949. Communication theory of secrecy systems. *The Bell system technical journal* 28, 4 (1949), 656–715.

[58] Roman Shor, Gala Yadgar, Wentao Huang, Eitan Yaakobi, and Jehoshua Bruck. 2018. How to Best Share a Big Secret. In *Proceedings of the 11th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '18)*. Association for Computing Machinery, New York, NY, USA, 76–88.

[59] Arun Subbiah and Douglas M. Blough. 2005. An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, USA) *(StorageSS '05)*. Association for Computing Machinery, New York, NY, USA, 84–93.

[60] Madiha Tabassum, Tomasz Kosinski, and Heather Richter Lipford. 2019. "I don't own the data": End User Perceptions of Smart Home Device Data Practices and Risks. In *Fifteenth symposium on usable privacy and security (SOUPS 2019)*. 435–450.

[61] TPC. 2023. TPC-C: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/

[62] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-Level Signatures for Smart Home Devices. *Proceedings of the 2020 Network and Distributed System Security (NDSS) Symposium* (February 2020).

[63] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. 2020. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 157–180.

[64] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. 2022. COBRA: Dynamic Proactive Secret Sharing for Confidential BFT Services. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1528–1528.

[65] Marcel Van Der Veldt. 2019. Hassio Add-ons: Auto backup.

[66] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[67] Vorion. 2020. Create Automated Backups Every Day.

[68] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. 2016. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 611–626.

[69] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. 2017. Splinter: Practical Private Queries on Public Data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 299–313.

[70] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. 2020. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 297–308.