End-to-End Encrypted Applications with Strong Consistency Under Byzantine Actors

Natalie Popescu
Princeton University
Princeton, New Jersey, USA
npopescu@princeton.edu

Shai Caspin
Princeton University
Princeton, New Jersey, USA
scaspin@princeton.edu

Leon Schuermann
Princeton University
Princeton, New Jersey, USA
lschuermann@princeton.edu

Jingyuan Chen
Princeton University
Princeton, New Jersey, USA
leocjy@princeton.edu

Amit Levy
Princeton University
Princeton, New Jersey, USA
aalevy@princeton.edu

***Abstract*—Existing end-to-end encrypted systems provide applications with strong privacy and integrity guarantees, but fully trust the server for consistency. However, many applications consider strong consistency a security property: a Byzantine server can corrupt application state by manipulating operation orders. We present SCUBA, an end-to-end encrypted framework for strongly-consistent applications in a Byzantine setting. At its core, the SCUBA protocol establishes strong privacy, integrity, and ordering guarantees that let applications achieve up to the strongest single- and multi-key consistency models known to date. SCUBA upholds all consistency guarantees in the face of a Byzantine server, and lets clients prove a server misbehaved to a third party. We build four previously-unsupported applications on top of the SCUBA key-value store, and show that SCUBA imposes minimal overheads and can support many applications at once. SCUBA enables entire classes of applications to benefit from end-to-end encryption.**

***Index Terms*—Security, Consistency, Privacy, Byzantine Actors, Transactions, End-to-End Encryption**

1. Introduction

End-to-end encryption is commonly how client-based applications communicate through potentially untrusted servers, while maintaining the secrecy and integrity of data. Typically, clients store data locally rather than on a central server, and replicate application-level operations to peers through the central server while encrypting messages end-to-end. Recently, this model has become particularly common in messaging applications [37], [41], [34], [43], [38], document collaboration [12], [19], and health tracking applications [32]. While such applications seldom implement business logic on the central server, the server fulfills important roles nonetheless: it routes messages between clients, queues messages for offline clients, and orders messages.

Today, applications must explicitly *trust* servers to provide this functionality correctly. While current protocols for secure end-to-end encrypted applications protect the secrecy and integrity of individual messages from a Byzantine server, they do not do the same for *consistency*. Consistency is a critical property for application correctness that limits the different orderings of operations clients see and, typically, requires that all clients see all relevant updates (at least eventually). In fact, most deployed end-to-end encrypted systems only provide very weak consistency [52], [15], [24], [46], [48], [47]. Some systems provide consistent message ordering in practice, but clients cannot validate consistency [19], or can only validate consistency for (but not across) individual data objects [12].

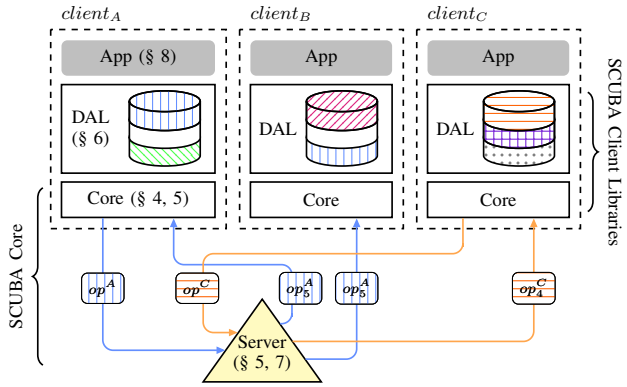


Figure 1: The SCUBA architecture. The server routes and orders messages between clients. The core layer handles encryption and message ordering. DALs let applications specify consistency requirements and data abstractions. Each client data store's colored pattern represents a unique set of data objects. $client_A$ and $client_C$ concurrently send messages that are totally ordered by the server: op^A and op^C are assigned sequence numbers 5 and 4, respectively.

Such weak guarantees may be sufficient for some applications—it may be acceptable for text messages in different groups to arrive in different relative orders. However, it is *insufficient* for other contexts and applications. For example, clients of a document collaboration application may see completely incompatible versions of a shared document if update messages are delivered in different orders. A health tracking application might render meaningless results if

some messages containing data updates are dropped. Worse yet, a secure messaging application may deliver a secret message to an outdated recipient if the operation changing group membership is ordered after the secret message.

Preventing a potentially Byzantine server from violating the consistency of private applications is challenging. First, in private multi-user applications, clients only see a subset of all operations and thus cannot validate a global total order independently. Second, practical applications must operate when many clients are offline, so traditional consensus algorithms are impractical. Third, this must be done efficiently on the client and with a practically scalable server. Finally, consistency validation should extend to complex applications that operate across objects shared by different groups, without leaking group membership. Preventing a malicious server from altering data consistency is particularly important for contexts in which data updates are causally related—removing a user from a group chat and then adding their sworn enemy, or updating a password and then using it.

We argue that addressing these challenges is tractable if servers are not arbitrarily malicious, but are unwilling to be caught misbehaving (§ 2). In particular, servers will not violate consistency if clients can quickly detect a consistency violation and efficiently prove the server is at fault to a third party (other clients which would cease using the server, or a regulatory body with oversight over the server operators).

This paper presents SCUBA—Strong Consistency Under Byzantine Actors. SCUBA (§ 3) is a protocol for consistent end-to-end encrypted applications that both detects and holds Byzantine servers accountable for consistency violations (the architecture is depicted in Figure 1). SCUBA is the first to provide these guarantees when clients may be offline or share objects across different, potentially overlapping groups of peers. SCUBA's core contributions are:

- A private multi-group transaction protocol (§ 4).
- A validation scheme letting clients detect server consistency violations and *prove* them to a third-party (§ 5).
- A client-side data abstraction layer over SCUBA that exposes a typical key-value store API to applications (§ 6).
- A server architecture and implementation that scales horizontally to service increasing message throughput (§ 7).

To evaluate SCUBA's generality we build four applications: a password manager, a family-based social media application, a healthcare calendar, and an auctioning application (§ 8). We evaluate SCUBA's overhead on clients (§ 9.2) and show that SCUBA has acceptable impact on memory consumption and application latency. We also evaluate the scalability of our server (§ 9.1) under a variety of workloads and show that it scales linearly in most cases by adding machines and achieves up to 800 000 delivered messages per second with 16 commodity machines. This shows that SCUBA provides significant assurances against Byzantine servers while being practical for even large applications.

2. Threat Model

Like other end-to-end encrypted systems, SCUBA models the server as malicious with respect to privacy and integrity. SCUBA also operates under a *covert* threat model for the server's ordering guarantees, in which the server may deviate from promised behavior, but does not wish to be caught doing so [3]. This fits a model of the world in which clients pay for routing their messages through a central service. Server operators are economically motivated to provide a functioning service, cannot gain information by peeking into messages, but *may be tempted to cut corners to decrease costs*. Reputation and profits are at stake if they are caught misbehaving. Thus, if clients can catch a misbehaving server and prove to other clients that the server did indeed misbehave, then server operators are disincentivised to misbehave. In contrast, a malicious threat model over server message ordering would severly limit application liveness and performance by requiring many round trips or long wait times for offline clients. Under the covert threat model, the server may try to read, modify, reorder, or drop messages, but is trusted not to completely fork clients (*i.e.,* the server will not prevent two clients from receiving any communication from one another or updates on any shared state, similar to a network partition). Thus, the SCUBA server is modeled as Byzantine with respect to message ordering and hence application consistency, untrusted with respect to privacy and integrity, and trusted with respect to general availability (and not forking clients).

SCUBA clients operate under a semi-trusted model with peers. Clients trust peers with the privacy of shared data, but those peers are Byzantine and may try to break consistency (*e.g.,* exclude a peer from an update) or access control rules.

SCUBA does not address orthogonal open problems in end-to-end encrypted applications, like key management or metadata attacks (traffic analysis). Like many deployed end-to-end encrypted applications, SCUBA assumes that network adversaries do not have global knowledge, the server discards message metadata, and users correctly manage their keys. SCUBA does not protect against Dolev-Yao attackers [11], but could be used to mitigate such attacks. SCUBA does not protect against other client-side attacks, like social engineering. Existing mitigations for the above attacks can be used in concert with SCUBA [5], [4], [9], [8], [50], [45].

3. Architecture

SCUBA is a client-based architecture in which clients both *store* and *coordinate updates to* all application state. Furthermore, SCUBA embraces a layered design (Figure 1). The SCUBA server routes and orders messages between clients. The core layer implements the SCUBA core protocol to enforce message secrecy, integrity, and ordering. Finally, the Data Abstraction Layer (DAL) lets applications specify their consistency requirements, application invariants, and access control policies. Various DALs can provide application developers with diverse data, storage, and consistency models. Applications are built on top of a DAL, such that developers only need to write application-specific logic, and otherwise have easy-to-use abstractions that conceal encryption, consistency, and underlying data store details.

layer	API	fields
core	→ send	`[(op, [rcpt])]`
	← receive	`op, sender, seq`
server	→ send	$(C_K(op)$, `[(rcpt,` $C_{rcpt}(K))]$)
	← receive	(C_{rcpt}, C_K), `sender, seq`
	→ delete	`seq`
	→ put	`otkey`

TABLE 1: APIs between SCUBA layers. DALs interact with the core API, and the core interacts with the server API.

A DAL interacts with the SCUBA core using the core API described in Table 1. The DAL sends the core a *series* (list) of messages, each message containing an operation and its recipients. All messages in a series are sequenced consecutively. Operations are reads or writes. The DAL sends one or more messages in each series, depending on the consistency model (described further in § 4). All messages received back by the DAL are ordered by the core layer. The DAL updates the store and application accordingly.

The SCUBA core is an end-to-end encrypted protocol that orders messages between clients according to a total, real-time order. SCUBA provides end-to-end encryption by authenticating and encrypting *all* messages exchanged between clients. SCUBA uses Signal's double ratchet scheme [25] with an optimization [22] that only (symmetrically) encrypts the operation once for all recipients, and then encrypts the symmetric key individually for each recipient using double-ratchet sessions. Figure 2 shows how outgoing operations are encrypted into these two parts: a common, authenticated ciphertext $C_K(op)$ and a per-recipient, authenticated ciphertext $C_{rcpt}(K)$. The encrypted message series are sent to the server, which is responsible for ordering them according to some global total order and delivering them to each recipient in that order.

The server must then route each message in the series to all its recipients according to the global total order. This protocol mandates a client addressing scheme, which in our case is provided by the public keys used by the underlying encryption protocol. To initiate communication, clients must first communicate out-of-band to exchange public keys, which in practice can happen through pre-existing channels (*e.g.,* text messages) or in-person via QR codes.

The server API, shown in Table 1, allows clients to *send* a series of messages to each other, *get* received operations, and *delete* (acknowledge) received operations. To support temporarily disconnected clients, received operations are cached at the server, stored in per-client inboxes, and deleted when acknowledged. The server also provides a number of endpoints for clients to exchange and update *one-time keys*, used by the double-ratchet encryption scheme to initiate pairwise, encrypted sessions. Finally, clients can choose to use a *server-sent events*-compatible [51] interface for streaming incoming operations from the server.

4. Strong Consistency Under Honest Actors

Applications require different consistency guarantees and abstractions over storage and data sharing: a file sharing application needs a linearizable data store organized as a hierarchical file system with access control lists, while an address book application can use a causally consistent key-value store. To this end, the SCUBA *core protocol* simply exposes an ordered log of operations reflecting the server-assigned ordering. A DAL or application can translate this to higher-level application semantics through a `receive` hook. This simple ordered log abstraction enables the straightforward construction of data stores with consistency guarantees as strong as Linearizability and Strict Serializability. Since these consistency models are impractical in many real world cases—requiring clients to be online to read or write any data—SCUBA also supports data stores with weaker but more performant consistency models.

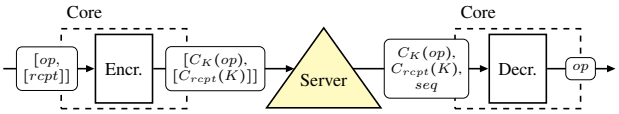


Figure 2: The path of an operation through the SCUBA core layer under the assumption of *non-Byzantine actors*. Outgoing operations are encrypted into two parts. Incoming operations include a server-assigned sequence number seq which dictates the operation's place in the global total order.

To explain how SCUBA reasons about consistency in the *non-Byzantine* case, we first map primitives in the SCUBA model to the established concepts of *shards* and *replicas*. We then describe how the core protocol's strong ordering properties can be used to construct a distributed data store supporting a range of single-key consistency models. Then, we describe how the core protocol supports multi-key transactions, and how that maps to two transactional consistency models. We also highlight the offline client behavior that each consistency model enables. Note that we define offline behavior with respect to *individual* clients, not groups—*e.g.,* if clients must be online to access a linearizable data store, a single offline client will not impact other clients' ability to access the store. These offline behaviors correspond to the DAL configuration flags we describe in § 6.

Application state, independent of its particular representation, can generally be decomposed into distinct data objects: a key-value pair in a key-value store, an entry in a database, or a file in a file system. Each such object can be read by many clients, *i.e.,* those clients hold a copy of the data locally. We define a *shard* in the global SCUBA data store as one or more objects shared with an identical set of clients. Each client is thus a *replica* of each shard it stores.

4.1. From Ordering to Consistency

To illustrate how SCUBA supports a range of consistency models, we now describe how the core protocol achieves two ordering properties that consistency models may rely on: a global total order and a real-time ordering constraint.

SCUBA uses its central server to establish a global total order over all operations. A client thus orders an operation on a data object by sending it through the server, as in Figure 2. Depending on the desired consistency model,

clients decide whether to order reads and writes. A client performing a *read* only sends that operation to itself, which will order the read with respect to the client's other operations. If reads are totally ordered, they are only serviced—and their result externalized to the application—once they are received back from the server. On the other hand, a *write* operation is broadcast to every single shard replica. If writes are totally ordered, they are applied on each replica—including on the issuing client—only and exactly when they are received back from the server.

Writes thus mandate that each client know and correctly specify which peers act as replicas for which shards. These mappings are dictated by the DAL or application, which determine which peers an operation should go to. Managing these mappings in a higher-level layer enables the core protocol to be agnostic to mappings and membership changes, merely enforcing that writes go to all specified peers. A scheme for tracking this information is described in § 6.3.

These semantics ensure that operations are processed on each client according to a global total order. A real-time order is guaranteed when SCUBA clients only have one outstanding operation (read *or* write) at a time. Awaiting the server-ordered response to every operation before (1) applying that operation and (2) sending the next operation, naturally ensures that operations respect a real-time order.

Linearizability. Operations on objects are linearizable if there exists a valid total order across all operations, *i.e.,* one that respects data dependencies and real-time constraints. Linearizability is thus upheld by clients sending all reads and writes to the server, and only dispatching one operation at a time. While Linearizability is a very strong programming abstraction, *a linearizable store requires clients to be online to perform read or write operations*. Applications can use weaker consistency models to maintain offline functionality.

Sequential Consistency. Sequential Consistency relaxes Linearizability's real-time ordering constraint. A sequentially consistent data store thus does not sequence read operations through the server, but must still only apply write operations once they are received back from the server. Consequently, *a sequentially consistent data store requires that clients be online to write data, but permits offline reads*.

Causal Consistency. Causal consistency only requires a partial order across operations and no real-time ordering constraint, thus write operations may be applied locally before they are sequenced by the server. A client sends writes to the server some time in the future in the same order as they were applied. Hence, *clients of a causally consistent data store can both read and write offline*.

4.2. Transactions

Multi-key atomic operations—or transactions—in SCUBA can span multiple shards replicated on clients with no pre-existing communication: in Figure 3, a transaction initiated by $coord_A$ includes data on both $client_A$ and $coord_B$, although $client_A$ and $coord_B$ share no data. To preserve client secrecy, the transaction must not leak information

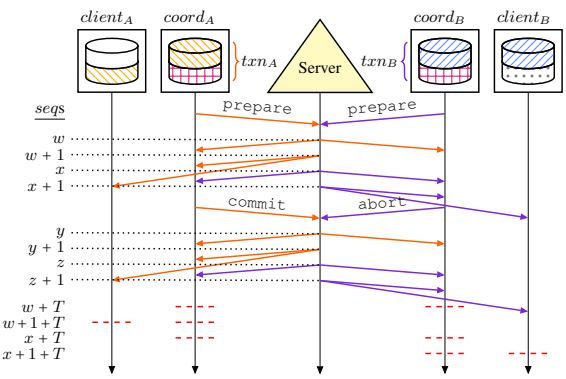


Figure 3: The SCUBA transaction protocol. txn_A and txn_B are concurrent and conflict on the pink, gridded shard. Each coordinator sends `prepare`s to all shards in their transaction. $coord_A$ receives it's own `prepare`s before any others, so it sends `commit`s to all txn_A shards. $coord_B$ receives a `prepare` from $coord_A$ before it's own, so it sends `abort`s to all txn_B shards. Recall, each message series is ordered consecutively. Timeouts are shown as dashed lines.

about any other shard's operations: $coord_A$ must not leak the operation on $client_A$'s shard to $coord_B$, or vice versa.

Naively, distributed transactions can use a two-phase commit protocol (2PC) to coordinate across clients with different views of the global data store. Yet, using 2PC in SCUBA's architecture reveals to each replica the recipients and operations involved on other shards: $coord_A$ would leak the operation on $client_A$'s shard to $coord_B$, and vice versa.

Transaction Protocol. SCUBA uses a simplified 2PC protocol to support privacy-preserving transactions. Internally, SCUBA separates a transaction's coordination messages to each set of shard replicas. The server's global total order implicitly determines winners between concurrent, conflicting transactions, eliminating the need for clients to vote. Message ordering in the prepare phase prevents clients from accepting conflicting transactions. The protocol also relies on the server's guarantee that each message series will be sequenced *consecutively*, making it impossible for another transaction to be sequenced during the prepare phase.

Figure 3 summarizes SCUBA's transaction protocol. Each coordinator first sends a `prepare` message series, with one message for each shard the transaction spans (including itself). Each coordinator then waits to receive all `prepare` messages back from the server. If no conflicting transactions are received before the coordinator receives its `prepare`s back, it sends a `commit` message series (see $coord_A$). If the coordinator receives at least one conflicting transaction's `prepare` before it receives its own `prepare`s back, it sends an `abort` message series (see $coord_B$). Notably, each shard replica receives independent `prepare`, `commit`, and `abort` messages and learns nothing about other operations or shards in the transaction.

The server's consecutive ordering guarantee is crucial for transaction isolation and atomicity. If `commit` messages interleave, clients would observe different serial executions of transactions, violating isolation. Consecutive sequencing

```
// Client-side data structures storing POVS
// history information
struct Histories {
  histories: HashMap<DeviceId, History>, }
struct History {
  history: VecDeque<HistoryEntry>,
  offset:  usize, }
struct HistoryEntry {
  digest: Digest,
  seq:    usize, }

// Message contents including validation payload
struct CommonPayload {
  op:     Operation,
  sender: DeviceId,
  rcpts:  Vec<DeviceId>, }   // added in Sec 5
struct PerRecipientPayload {
  symm_key: Key,
  head:     Digest,          // added in Sec 5
  index:    usize, }         // added in Sec 5

// Check validity of validation payload
// included in incoming message
fn validate_payload(self,
    common: CommonPayload,
    per_rcpt: PerRecipientPayload) -> bool {
  let history = self.histories[common.sender];
  return history[per_rcpt.index] &&
    history[per_rcpt.index] == per_rcpt.head; }

// Generate new entries in histories for peers
// included in an incoming message
fn add_to_histories(self,
    common_ciphertext: &[u8],
    common: CommonPayload) {
  for peer in common.rcpts {
    let history = self.histories[peer];
    history.append(hash(
      common_ciphertext + common.rcpts
      + history.head())); } }
```

Listing 1: The data structures and functions for validating and updating POVS state.

also helps all transaction participants unanimously commit or abort any conflicting transactions, crucial for atomicity.

To ensure liveness, each transaction has a relative timeout T, a sequence number offset from each `prepare` message. If clients do not receive a `commit` message by T, they locally abort. For timeouts to be uniform across different shards, each coordinator's `prepare` and `commit` message series must list all shards in the same order; otherwise, a `commit` that races with the timeout can cause some shards to commit but not others. Due to consecutive sequencing, if a coordinator's `commits` race with the timeout, they will always be ordered either before or after the timeout.

Strict Serializability. Strict Serializability is a consistency model that requires all data operations (including reads) to be part of transactions. It further requires that transactions respect real-time ordering. *This is achieved by each client only coordinating one transaction at a time.*

Serializability. In contrast, Serializability does not require real-time ordering. Thus, if a transaction only contains read operations, it does not need to be sent through the server. Hence, *Serializability admits offline reads*.

Client ID	Previous seq	Next seq
seq_a	$H(C_K(common)_a)$, $[(H(rcpt \Vdash seq_a), H(C_{rcpt}(per\_rcpt)_a \vee \mathbf{0}))]$	
$\vdots$	$\vdots$	
seq_n	$H(C_K(common)_n)$, $[(H(rcpt \Vdash seq_n), H(C_{rcpt}(per\_rcpt)_n \vee \mathbf{0}))]$	

Figure 4: A *server attestation* is an Ed25519-signed digest closing over the outlined fields. H is a cryptographic hash function. The attestation spans all operations delivered to the indicated client within the range [Previous seq; Next seq). Each client's attestations combined cover the entire contiguous sequence space. Thus an attestation's Previous seq field must be equal to the preceding attestation's Next seq field. Each row in an on-send attestation represents a *promise* of a sequenced message, while each row in an on-receive attestion represents a sequenced message. Subscripts a and n denote fields associated with messages a and n, respectively.

5. Strong Consistency Under Byzantine Actors

SCUBA enables strong consistency guarantees that no other end-to-end encrypted systems currently ensure. As described so far, SCUBA only provides these guarantees if all clients and the server are honest. Yet a Byzantine server may misbehave by dropping messages to some or all clients, assigning different sequence numbers to the same message, or ordering series non-consecutively. Byzantine clients may not provide full recipients lists to the server, or not encrypt a message correctly for a subset of clients. Any such misbehavior violates consistency by causing shard replicas to diverge.

Pairwise Order Validation Scheme (POVS) is a client-side mechanism that allows clients to detect when shards diverge, and prove—when the server is at fault—that the server caused the inconsistency by misbehaving. The proof of server misbehavior is efficiently constructed, efficiently verifiable by any third party, and leaks only limited metadata. As clients can construct proofs for any server misbehavior, servers will behave under the covert threat model.

5.1. Pairwise Order Validation Scheme

At a high level, each client stores a condensed history of its communication with its peers in a simple data structure we call the *hash-chain-history* (HCH). By comparing HCHs, peers can confirm that they have seen the same messages in the same order. The server provides an *attestation* for every message sent and received, vouching for the message's ordering and contents. If two clients' HCHs differ, they can use the server's attestations to prove server misbehavior.

While a hash-chain is a single cryptographic digest closing over operation data and the prior hash-chain value, an HCH is an ordered list comprising all prior states of a hash-chain. Each client stores an HCH for each peer it communicates with. An HCH's latest entry is the *head*. Each entry consists of a hash-chain digest and the sequence number for the corresponding message (lines 3-10 of Listing 1).

A cryptographic server attestation is shown in Figure 4. For a range of sequence numbers, the server attests (via

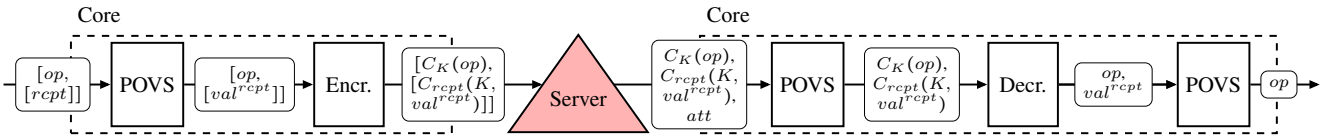


Figure 5: The path of an operation through the SCUBA core augmented with mechanisms for mitigating Byzantine behavior. POVS forwards the per-recipient validation payloads $[val^{rcpt}]$ with the plaintext operation op. Encryption then creates a common ciphertext $C_K(op)$ from the recipients list (omitted) and op, and a per-recipient ciphertext $C_{rcpt}(K, val^{rcpt})$ from the symmetric key and each recipient's validation payload. The server piggybacks its on-receive attestation att onto each recipient's ciphertext. Each recipient stores att, decrypts the ciphertexts to get op, val^{rcpt}, and the recipients list, and validates the payload. If no misbehavior is detected, all recipient HCHs are advanced and op is forwarded to the next layer.

public-key signature) that a client has sent or received an exact set of operations. Each message is represented by a sequence number, a digest of the common ciphertext, and, for each recipient, a tuple closing over the recipient's identifier (concatenated with the assigned sequence number) and per-recipient ciphertext. Clients receive attestations for messages they send (*on-send*) and receive (*on-receive*). On-receive attestations only have the reciever's per-recipient ciphertext (all others are 0), while on-send attestations close over all recipients' ciphertexts. This hashing scheme allows clients to prove misbehavior by revealing the recipient identifier and the common digest for the problematic message.

Message Path. Figure 5 shows the path of operations through the SCUBA core with additional POVS stages (compare to Figure 2). When a client sends a message, it now adds a *validation payload*: each peer's respective HCH head value and index, as shown on lines 19 and 20 of Listing 1. The head encapsulates up to the last message that *both* the sender and recipient should have seen.

Once the client sends the message series to the server, the server replies to the sender with an on-send attestation testifying that it *will* deliver the messages unmodified, to all recipients, and with a specific, consecutive set of sequence numbers. When the server delivers each individual message to a client, it includes the full sender-provided recipients list, as well as the on-receive attestation testifying to the message's sequence number and recipients list.

When a client receives a message, it first checks attestation validity and stores it in case a proof must later be constructed. Clients check attestation validity by reproducing the hashes included in the attestation based on the server-provided sequence number, recipients list and ciphertext. After decryption, the client compares the validation payload in the message with its own local HCH, as shown on lines 24-29 of Listing 1. If validation succeeds, the recipient knows it saw all messages up to and including the just-validated message in the same order as the sender. POVS then advances the local HCHs of *all* recipients in the sender-provided recipients list, including that of the sender, as shown on lines 33-40 in Listing 1. POVS uses SHA-256 as its cryptographic hash function.

Optimizations. To optimize storage, POVS can trim each peer's HCH once it is validated. For simplicity, we assume that the validation payload is sent with every message, so recipients would trim the sender's HCH on each received message. Applications may configure the frequency with which validation payloads are sent. An attestation can be discarded once all HCHs do not contain any unvalidated entries for messages with the sequence numbers covered by the attestation. Applications that expect clients to be offline for long periods of time could implement a membership pausing scheme in the DAL (*e.g.,* by using transactions over group changes or storing a list of offline clients) to temporarily enable history trimming.

5.2. Detecting Misbehavior

POVS lets clients detect consistency violations by a Byzantine server or client. If the server misbehaves, POVS lets clients prove this with attestations. If a client misbehaves, other clients can detect (but not prove) it. A proof of server misbehavior can be constructed *if and only if* the server misbehaved; no client misbehavior leads to a proof of server misbehavior. In practice, this means a client observing misbehavior with respect to a peer can assume the peer misbehaved if they cannot cooperate to construct a proof.

In general, a server can misbehave by dropping, modifying, or reordering a message for a subset of its recipients. It may also generate an incorrect attestation. On the other hand, a client can improperly encrypt a message to a subset of recipients or attach an incorrect validation payload.

Three types of misbehavior are immediately detected upon message reception. If the server provides an incorrect attestation, validation of the attestation's hash or public-key signature will fail. If the server violates its consecutive ordering guarantee for series, the sender will observe this in its on-send attestation. If the server provides a different recipients list than that in the encrypted common payload, both the sender and receivers can detect this in on-send or on-receive attestations, respectively. Detecting remaining misbehavior requires active communication between clients.

Clients detect remaining violations by comparing POVS validation payloads against local HCHs. Either the validation payload's index is out of range, or the digest at that index does not match the digest in the validation payload. Clients thus detect misbehavior on the *next* validation payload exchanged between two clients with diverging views.

Byzantine Server. From the perspective of two message recipients, the server may misbehave (with respect to a single message) in the following ways: fail to deliver the

message to one of the recipients; deliver the message at the same sequence number but with different recipients lists, per-recipient ciphertexts, or common ciphertexts; or deliver the message out-of-order by changing the sequence number for one or both of the recipients. If the validation payload index is out of range, then the receiving client did not receive a message that another client received before constructing the validation payload, *i.e.,* the server dropped a message. All other misbehaviors (modifying or reordering messages) manifest in the validation payload digest not matching the local digest at the validation payload index. When a client detects this, it attempts to prove the misbehavior occurred by exchanging and combining attestations with the peer that sent the validation payload. We describe this further in § 5.3.

Byzantine Clients. A malicious client could encrypt a ciphertext incorrectly, provide an incorrect recipients list to the server, or attach an incorrect validation payload for one or more recipients. If a sending client does not faithfully list all recipients, then receivers' attestation validation will fail—the reproduced hashes will not match those in the on-receive attestation. If a sending client sends an incorrect validation payload, then the recipient's POVS validation will fail. If a sending client does not encrypt a common payload correctly, then decryption fails.

In all cases, the receiving client will first try to prove server misbehavior. If they fail to prove server misbehavior because the peer with the necessary attestation does not provide it, then the client will declare the peer misbehaved. For example, if a client cannot decrypt a message, they will ask the sender for their on-send attestation so they can compare their per-recipient and common ciphertexts. If the sender provides an attestation, either the attestation matches and the sender is at fault, or it differs and a proof can be constructed against the server. If the sender does not provide an attestation, they can be deemed uncooperative and therefore malicious.

Client-Server Collusion. If a client colludes with the server to make another client inconsistent, the inconsistency can be detected by any two non-colluding clients that witness diverging messages as a result of the collusion. Only *two* honest clients with divergent histories are ever required to detect any misbehavior.

5.3. Proving Server Misbehavior

For any server misbehavior, there will exist an attestation on at least two clients that cryptographically proves the server misbehaved. Clients find these attestations by comparing HCHs and finding the sequence number of the first message that leads to a divergence. Depending on the misbehavior, the attestations corresponding to this sequence number, from two clients, can be combined in one of two ways.

Matching Sequence Numbers. If both clients hold an attestation with an entry for the problematic sequence number, then either different message contents or recipients causes the divergence—at least one hash will differ between the two attestations. In this case, the server signed conflicting information which is sufficient to prove misbehavior.

If a client cannot decrypt a message, its per-recipient ciphertext may have been modified by the server. Since each client is only exposed to its own per-recipient ciphertext, an on-send attestation, containing another instance of this recipient's per-recipient ciphertext digest, is required for this proof. If the per-recipient ciphertexts differ, then the two attestations will show different hashes for those fields. Similarly, the server thus signed over conflicting information.

Missing Sequence Numbers. Dropped or reordered messages cause recipients to see inconsistent sequence numbers: one peer will hold an attestation for a message at a specific sequence number, which the other did not receive. As attestations cover the entire contiguous sequence space through the Previous and Next seq fields, the other peer will hold an attestation covering the sequence number of this message. Thus both clients hold attestations for the sequence number: one including the problematic message, and the other missing it. This is an indication of misbehavior only if the client lacking this message is listed as a recipient in the attestation with the message. This client's identifier and the digest of the common ciphertext hence need to be unsealed in order for a third party to verify this proof.

5.4. Recovering From Server Misbehavior

Based on the consistency model, data types, number of writers, or frequency of updates, different recovery methodologies may be ideal. For example, in a healthcare calendering application, stepping into a stop-the-world state after misbehavior is detected and resolving the inconsistencies by hand may be preferable. Alternatively, an OT-style application would benefit from rollback-based recovery, or an infrequently-updated application might favor snapshot-based recovery. Thus, SCUBA designates the DAL or application layer to adopt the most appropriate recovery solution.

6. TANK: A SCUBA Data Abstraction Layer

A Data Abstraction Layer (DAL) is a wrapper around an underlying data store. It propagates, checks, and applies operations on the data store. A DAL operates on arbitrary data types, and performs application invariant and access control checks, as shown in Figure 6. This paper presents TANK (yeT ANother Key-value store), an implementation of a DAL. TANK is written in 4000 lines of Rust.

6.1. The TANK Data Model

TANK wraps a key-value store, but DALs could wrap any data stores, such as relational databases. TANK allows applications to configure the consistency model through flags on instantiation. These flags dictate whether or not to (1) synchronize reads through the server, (2) block on receiving write acknowledgments back from the server, and (3) enable multikey transactions. Note, these flags only affect client

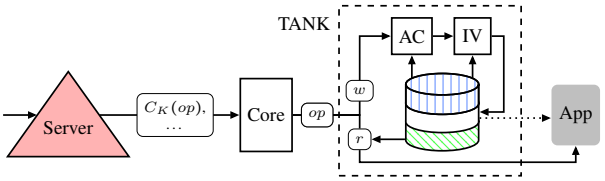


Figure 6: Interactions within TANK as *op* is *received*. TANK differentiates between reads r and writes w. A read is serviced using the current state of the store. The result is sent to the application. A write undergoes access control (AC) and invariant validation (IV) checks before being applied.

behavior. The *synchronize reads* flag configures the client to (when true) send all reads through the server to sequence them in the total order, or (when false) perform reads locally. The *block on write acknowledgments* flag configures the client to (when true) only perform a subsequent read or write once the current write has been sent to the server and the server's acknowledgment is received back by the sending client (*i.e.,* once the write has been sequenced by the server), or (when false) permit subsequent reads and writes even though previous write acknowledgment(s) have not yet been received back. Finally, the *multikey transactions* flag configures the transaction coordinator. A sequentially consistent data store should be configured with (1) false, (2) true, and (3) false, while a serializable data store should be configured with (1) false, (2) true, and (3) true.

Data Types. TANK internally operates over a generic data type that wraps all application-specific types. In particular, TANK's generic data type has the following fields: `id`, `value`, `type`, and `perm_id`. The `id` is a unique, application-specified string. `value` holds the application-specific object, which can have any structure. `type` is an application-specific identifier that associates an object with invariants (described in § 6.2). Finally, `perm_id` is a pointer to a key associated with a permissions object, used for access control (see § 6.3).

Application API. TANK offers a `get(key)` and `set(key, value)` API to applications. It also exposes an `add_perm(key, [clients])` API for permissions types (§ 6.3), enabling sharing data with permissions.

If TANK is configured with transactions, it exposes `start_tx` and `end_tx` operations, which internally queue single-key operations. Aborted transactions are not automatically retried. The value for transaction timeout T is set and updated by TANK, and is assigned dynamically based on server load. T is the difference between the previous transaction's `commit` and `prepare` messages' sequence numbers, multiplied by three. Different timeout mechanisms could also be used (*e.g.,* TCP-like window scaling). In the future SCUBA could let applications specify timeout mechanisms based on expected application usage.

6.2. Application Invariants

TANK lets developers register invariant-checking functions for specific data types or all writes; *e.g.,* a social media application may limit the number of characters allowed in a post or comment. These functions run before a write is applied to the data store (Figure 6). Invariant-checking functions only run on writes, as reads cannot violate invariants. TANK could be optimized to also run checks on outgoing writes to prevent sending invalid operations.

6.3. Access Control

Since data is replicated across all clients that can view it, clients with read but not write permissions may attempt to modify data. TANK uses *permissions* types mapped to client *groups* for access control. Access control checks are only applied when a write is received by a client. Since clients implicitly have read permissions for all shards they replicate, reads are not checked.

Groups organize clients into lists of groups or client identifiers. Groups are n-ary trees, enabling applications to specify arbitrary sharing policies. A *user* is a group containing multiple client identifiers, typically associated with devices. Additional groups, *e.g.,* a family, can then be constructed to include multiple users. Each group is resolved into a list of clients by recursively traversing subgroups.

Permissions map one of several privilege types to a group. Every group and data object is guarded by a permissions object. Applications can extend TANK with new privilege types, but TANK provides four default types: owner, admin, writer, and reader. All privilege types can read both the data and metadata objects. Writers can modify data but not metadata. Owners and admins can change object metadata, *e.g.,* adding a clients to the writer group of a permissions object. Admins can never remove an owner's privilege.

7. A Scalable Server

The SCUBA client-server interface is amenable to different server architectures and implementations. We present a design for a *scalable* sharded architecture based on hierarchical sequencing, as explored in Scalog [10], and sharded message routing, as commonly implemented in publish/subscribe message brokers such as Redis [27]. The design is depicted in Figure 7. The server is written in 2700 lines of Rust.

The server processes messages in *epochs* coordinated by a central sequencer. Clients are assigned to particular server shards and corresponding outbox (`OB`)- and inbox (`IB`)-actors on their shard through a digest of their public key. Clients send message series to outbox actors on their designated shard. Messages are assigned hierarchical sequence numbers at the outbox actor by concatenating, in order, the *epoch*, *shardID*, *outboxID*, and *queueIndex*. The outbox actor then breaks up the set of per-recipient ciphertexts and sends each, with the common ciphertext, to the inter-shard router (`RT`) for the recipient. Inter-shard routers aggregate messages from each outbox actor on the shard. When an inter-shard router has received all messages, it sends them to the appropriate inter-shard receiver on the remote shard via TCP/IP. Inter-shard receivers break up the

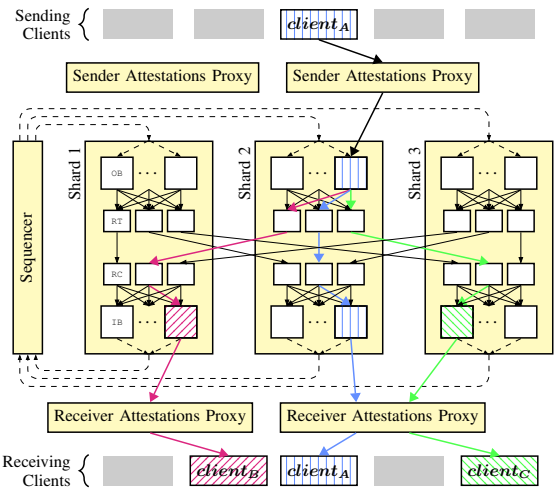


Figure 7: Architecture for an actor-based SCUBA server. The server has a sequencer and multiple shards. Each shard is composed of multiple client outboxes and inboxes, and internal routers and receivers for cross-shard communication. In this figure, $client_A$ sends a message to $client_A$, $client_B$, and $client_C$, which is routed across server shards.

collection of messages and forward them to each recipient's inbox actor. Inbox actors aggregate messages from each inter-shard receiver. Messages are exposed to recipients once all are received by inbox actors.

Epochs are coordinated by outbox and inbox actors. When inbox actors finish collecting messages, they send an *epochComplete* message to the sequencer, which awaits *epochComplete* messages from all shards. Once received, the sequencer initiates a new epoch and notifies all shards, which notify their local outbox actors. A new epoch causes outboxes to atomically process messages of the previous epoch, while accepting new messages in a separate queue.

This design requires no locking or sorting: shards and outbox-actors are assigned monotonically increasing IDs, making up the most-significant bits of the sequence space within an epoch. It is only required that inter-shard routers and inbox actors aggregate messages in their ID order.

The latency of an epoch is governed by the slowest inbox to outbox path. To avoid placing undue overheads on this latency-critical path, we generate both on-send and on-receive attestations on proxy servers. Generating an attestation involves hashing the messages' contents and generating an Ed25519 public-key signature. The server persists the sequence number of the last message received by a client so the attestation provided can close over it. These proxies require no coordination and can be scaled independently of other server components.

8. Applications

We build four previously-unsupported end-to-end encrypted applications with various consistency, data model, and access control requirements, demonstrating SCUBA's generality and expressivity. All are Rust command-line applications written in roughly 800 LoC.

Password Manager. Password managers store sensitive information: account passwords and two-factor authentication (2FA) shared secrets (TOTP or HOTP keys). The SCUBA password manager generates passwords and synchronizes them across devices. Each password type includes a username, password, application-distributed secret, and counter in the case of HOTP. Counter synchronization is trivial in SCUBA. The device on which the password is generated, *e.g.,* the owner, can grant peers writer permissions for reading and updating, or admin permissions to allow sharing the password with additional devices.

The password manager uses *Linearizable TANK* to order password changes according to a total order and real-time data-dependencies, so clients must be online to use the application. HOTP requests always use and increment the latest counter value, password reads always return the latest password, and password updates always propagate to the latest member list.

Family Social Media. Social media applications store sensitive user data such as posts, comments, locations, etc. Causal consistency is sufficient, as it, *e.g.,* ensures that a comment does not reference a post before the post is available. This allows clients to read and write data offline.

The family social media application shares updates across a family. Users can belong to several families. Family members can post, comment, and share live locations. Locations can be shared to subsets of members, but posts and comments are readable to the entire family. The family initializer grants members write or admin permissions for a family object. Posts and comments can be modified by their creators and have an invariant for maximum characters. The family application uses *Causal TANK*.

Healthcare Calendar. Healthcare calendars store private patient-provider data (purpose or frequency of visits) and must comply with healthcare privacy standards [1]. Calendering applications should allow patients to schedule appointments with providers without leaking private data.

The SCUBA calendar allows patients to request appointment times from providers. Providers confirm or deny appointment requests. An appointment object is private to the patient and provider. Providers share their availability with all patients via a specialized *metadata-private reader* permission type. This lets clients read object data but not metadata (*e.g.,* other readers), which in SCUBA means that object metadata is not sent to these clients. Patients receive updates to a provider's availability without knowing about other patients. To do this, the provider sends separate updates to each metadata-private reader.

When a patient requests an appointment, they grant the provider write privileges to the appointment object. When the provider confirms the appointment, their client *atomically*: updates the appointment status (sent to the requesting patient), and adds a new occupied slot to their availability (sent to all patients). The calendaring application uses *Serializable TANK* for privacy-preserving transactions.

Online Auctioning. Online auctioning applications have monetary incentives to cheat. They must enforce fair auc-

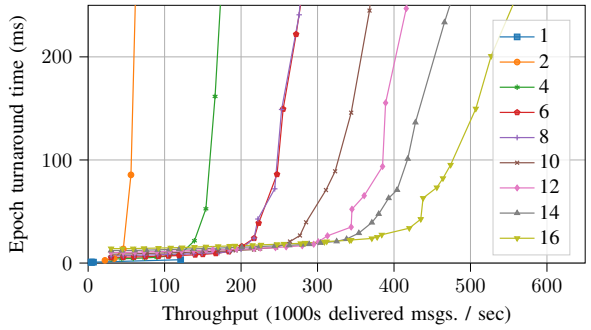


Figure 8: Server throughput vs. latency of the SCUBA server given different shard counts (1-16). Messages (1 kB common, 267 B per-recipient ciphertexts) are sent to 4 recipients, uniformly distributed across inboxes and outboxes.

tions. Namely, the auctioneer and all buyers must be notified of all bids, and that bids are publicized to everyone in the same order. Auctions are a prime example of how consistency violations can affect overall application security.

The SCUBA online auctioning application runs on *Sequential TANK* and supports both open and closed auctions. The seller, or another trusted client, acts as an auctioneer; they open the auction, accept and propagate bid announcements, and announce the final sale. The auctioneer creates the auction on behalf of the seller. In an open auction, the auctioneer announces the auction to all registered clients and accepts bids from any client. In a closed auction, the auctioneer communicates with a set group. In either case, the auctioneer gives participants read permissions to the auction object. To place a bid, clients create and share bid objects with the auctioneer, which only accepts the bid and updates the current highest bid value if the new bid arrives within the auction time and is higher than the current highest bid. Auctioning uses POVS to prove a bid was placed in a particular, consistent order. Auditors can ensure that the auctioneer announces all bids, accepts bids in the correct order, and choses the winner faithfully. If a bidder blames the auctioneer of interference, the auctioneer can use attestations to prove the server is at fault. Otherwise, clients can conclude the auctioneer is dishonest.

9. Performance

We empirically evaluate the SCUBA prototype in two aspects. First, we measure the latency given throughput of the server implementation, varying the number of server shards, message sizes, and number and overlap of message recipients. These results show the SCUBA server's ability to support many applications simultaneously. Secondly, we show that client-side overheads are minimal, and contextualize these overheads in terms of real-world applications.

9.1. Server Benchmarks

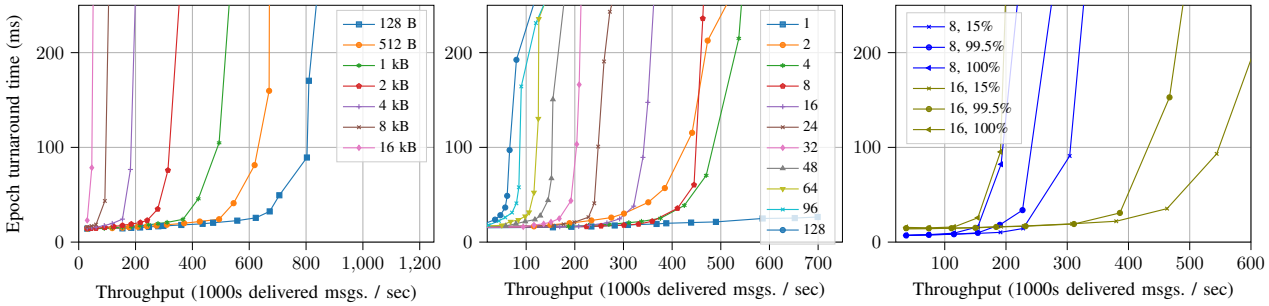
We evaluate the server described in § 7 using a heterogeneous set of 49 servers on the CloudLab Utah testbed. We dedicate one `c6525-25g` machine to the central sequencer, 16 `c6525-100g` machines to server shards, 16 `c6525-25g` machines to load-generating clients, and 16 `m510` machines to receiving and deleting messages from server inboxes, and generating attestations. Each client machine runs 32 closed-loop clients, one per hardware thread. Clients are distributed evenly across shards. This setup simulates 512 constantly sending clients, comparable to a setting with orders of magnitude more clients sending and receiving less frequently. As messages may be sent to multiple recipients, we measure throughput as the number of *delivered* messages. Latency is measured using epoch turnaround times. The latency of a message is at most the length of the current and next epochs. We evaluate the performance impact of four variables: shard count, message size, recipient count, and recipient inbox contention (Figures 8 and 9). In all experiments, we impose varying loads by throttling client request rates and measuring mean effective server throughput and mean epoch turnaround latency.

Scalability by Sharding. Figure 8 shows a throughput-latency plot with uniform load and different shard counts. Senders and recipients are evenly distributed over all shards. Each message is sent to 4 recipients. The server throughput scales in the number of shards: while using 4 server shards delivers approx. 175 000 msgs/s at 150 ms latency, using 16 shards delivers over 500 000 msgs/s at the same latency.

Message Size. Figure 9a shows the throughput-latency impact of larger common ciphertext sizes. The server's outbox actors must copy the common ciphertext for all recipients and deliver them to inbox actors, which may be located on different shards. Consequently, server performance is sensitive to the message size: while a 16 shard configuration can deliver approx. 800 000 msgs/s at a common ciphertext size of 128 B (with each message sent to 4 recipients and having 267 B per-recipient ciphertexts), this number decreases to approx. 200 000 msgs/s with 4 kB common ciphertexts.

Recipient Count. Figure 9b shows the throughput-latency impact of larger recipient counts. An increase in the number of recipients correlates to an increase in server load; since each message must be delivered to all recipients, larger recipients lists cause the server to process a higher number of *delivered* messages per epoch. This implies a fundamental limit in the types of applications SCUBA is suited for; namely, SCUBA is better suited for applications in which data is shared amongst a relatively small set of clients. Nonetheless, even for 32 recipients per message, latency remains acceptable under high throughput, with approx. 200 000 msgs/s at 100 ms.

Recipient Overlap. Finally, we evaluate contention when many messages are delivered to a small set of recipients. Figure 9c outlines various degrees of *inbox overlap*. Each curve shows a constant number of clients sending messages to a constant number of recipients. Each message's recipients list (size 4) is chosen from an increasingly small pool of total recipients. In the 0% overlap case (not pictured), each of the 512 sending clients send messages to a distinct set of 4 clients (of 2048 total receiving clients). In the 100% overlap case, all 512 sending clients share an identical set of 4 recipients. Senders and recipients are uniformly distributed



(a) Varying message size. (b) Varying recipient counts. (c) Varying shards and recipient overlap.

Figure 9: Throughput vs. latency of the SCUBA server under varying load. Insets (a) and (b) use a 16 shard configuration and distribute senders and recipients uniformly across shards. The message is a 1 kB common ciphertext and 267 B per-recipient ciphertext for insets (b) and (c). Each message in insets (a) and (c) is sent to 4 recipients. Even with message sizes in the kBs, group sizes greater than 100, or high contention, SCUBA handles hundreds of thousands of msgs/s.

across all shards. Server load increases with a higher degree of overlap, since each recipient inbox maps to a single thread on a single server shard. For 100% overlap, the server is effectively bottlenecked by the single-thread performance of a single server shard. Yet, for less skewed workloads (overlap ratios of 15% and 99.5%), increasing the number of server shards still corresponds to an increase in throughput.

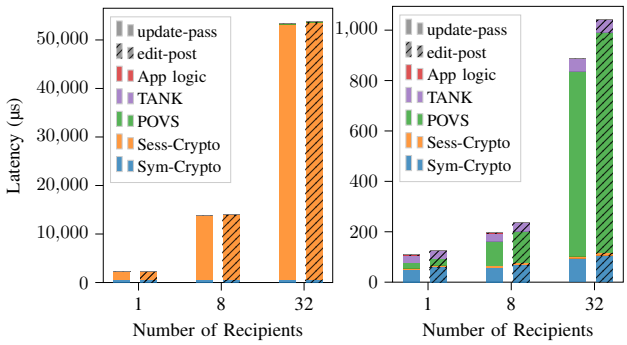
9.2. Client-Side Overheads

We measure client-side overheads on a CloudLab Utah `m400` aarch64 machine with 8 cores and 64 GB of RAM. SCUBA uses the Olm encryption library [23] as its double-ratchet implementation and AES-256-GCM for symmetric encryption. POVS uses SHA-256 as its hash function. The SCUBA core client library is written in 1900 lines of Rust.

Storage. Each client stores 328 B for identity and encryption keys. A one-time key pair is 328 B. An encrypted pairwise session requires approx. 1 kB. An HCH entry is 40 B (a 32 B digest and an 8 B index). Each client also stores two 8 B integers for the last-validated and trimmed entries. Each message in an attestation is 80 B with 128 B of metadata.

We now contextualize overheads. In a password manager client with eight peers, each password is shared with three peers that can all modify it, so the number of unvalidated entries is low (say two). Since password changes are rare, the number of attestations is low (*e.g.,* two) with one message in each. This amounts to just over 1.1 kB. In another scenario, an active family social media member has 20 peers, 50 attestations, and 250 operations. Other family members are less active, so the client has 50 unvalidated entries in each HCH (it is not receiving validations). This client needs 84 kB for storage. Finally, an auctioneer for open auctions may have 1000 peers, a small number of unvalidated entries (*e.g.,* 20) since the auctioneer is both sending and receiving (validating) payloads, 200 attestations, and 10 000 operations. The auctioneer needs about 1.6 MB of storage.

Bandwidth. Operation sizes are highly application-specific. In the password manager, initializing a 10 B password, a 20 B 2FA key, and a 14 B username, along with metadata like `id` and `type`, yields an operation size of 359 B.



(a) Sender latency, dominated by double-ratchet encryption. (b) Receiver latency, dominated by POVS validation.

Figure 10: Client-side latency for two application-level operations: `update_pass` operations are 359 B, and `edit_post` operations are 756 B. Client-side latency spans five categories: App logic = application code; TANK = access control, invariant validation, group resolution; POVS = HCH piggybacking and validation; Sess-Crypto = double-ratchet encr./decr.; and Sym-Crypto = symmetric encr./decr.

AES-256-GCM does not add bandwidth to the common payload. Each per-recipient payload adds 103 B. Of this, the common ciphertext's symmetric key is 32 B, the nonce is 12 B, and the authentication tag is 16 B. The remaining 43 B belong to the validation payload: the head entry (32 B) wrapped in a Rust Option (1 B), the head index (8 B) wrapped in a Rust Option (1 B), and a boolean (1 B) for sending dummy payloads when the sender is also the recipient. Because validation payload sizes are constant, the double-ratchet scheme always results in an encrypted per-recipient payload of 256 B, except when two clients communicate for the first time. These messages are 332 B, as Olm adds session-initialization metadata.

Computational. We now show the client-side latencies of two application-level *set* operations, which exercise validation payload generation, receiver validation, HCH advancement, encryption, and decryption. The password manager's `update_password` and the family social media applica-

tion's `edit_post` operations are 359 B and 756 B, respectively. We measure latencies for 1, 8, and 32 recipients.

We present results in Figure 10. Symmetric encryption and decryption overheads are proportional to common payload sizes. Double-ratchet encryption dominates sender overhead, is propertional to the number of recipients, and is constant on the receiving end (it only happens once). POVS validation payload generation is also proportional to the number of recipients, but overheads are low since sender-side POVS largely copies HCH entries. Receiver-side POVS (advancing HCHs) has the highest latency, but is still orders of magnitude lower than the off-the-shelf encryption performed by message senders. Notably, these numbers are negligible in comparison to network and the server latencies. Applications built on SCUBA can similarly expect latency to depend on message size and recipients list lengths, in addition to application-specific overheads.

9.3. Target Applications

We show that the SCUBA server scales horizontally in throughput and latency with additional shards. The server performs much better for smaller group sizes and messages (up to 1 kB), but still handles message sizes up to 16 kBs and group sizes in the hundreds. Client overheads are primarily dictated by message size and recipient count, but are reasonable compared to network and server latencies. Some applications with very large group sizes, large data mutations, and very high overall throughput are less appropriate for SCUBA, but *many* applications are well within bounds of a few dozen users per group and modest data mutation sizes. SCUBA is thus best-suited for privacy-oriented applications in which data is user-generated (health logs, notes), data is shared among few devices or users, and large data is not stored or sent (search engines, public social media).

10. Related Work

Application frameworks. Existing platforms and frameworks ease the burden of application development and deployment. Firebase [31] lets developers avoid building centralized application infrastructure, handling authentication, access control, and consistency in a centralized and general manner. Architectures like Jamstack [6], Solid [28], and Blockstack [2] envision application models that diverge from the traditional centralized architecture in favor of security. Similarly, SCUBA uses an application-agnostic server and exposes a set of simple and familiar developer APIs.

Private systems. Ghostor [16] lets clients detect a malicious central storage server wth blockchain-backed checkpoints. Other end-to-end encrypted applications for chatting, video conferencing, email, or file sharing [33], [39], [13], [40], [36], [35], [42] have weak consistency requirements and may store data centrally. Centralized systems [18], [20], [26], [14] or decentralized data stores [44], [12] provide weak consistency guarantees or weak access control rules. They do not handle Byzantine faults, and rely on a central server for storage, computation, or verification.

Distributed Consistency. Attaining strong consistency in a distributed setting is an age-old problem. Timestamping, vector clocks, and matrix clocks achieve Causal Consistency in a multi-processor setting. Distributed consensus [17] and multicast [7] protocols strictly define guarantees achieved by multi-roundtrip communication schemes. Protocols for State Machine Replication explore consistency in a fault-tolerant setting. Prior work also explores hash-chain based ordering: SPORC [12] uses hash-chains to validate a consistent ordering of operations on a single document, and Timeweave [21] uses hash-chains to establish a partial order over distributed histories. Blockchain has provided ordering and consistency over centralized [16] and decentralized [12] systems, often with high overheads. POVS, however, validates a consistent total order across pairs of clients for all their observable operations, thus validating order across shared objects.

Secure Messaging Guarantees. Secure messaging guarantees are crucial, but do not generalize to broader applications. A Secure Messaging SoK [49] describes a *causality preserving* property that is insufficient for applications with stronger consistency requirements (*e.g.,* Linearizability or Sequential Consistency). SYM-GOTR [30] establishes *global transcripts*—an agreed-upon ordering of all messages within a group of participants—in a *synchronous* setting: all clients need to approve a message before it is exposed, which is prohibitive in the face of offline clients. Mobile CoWPI [29] presents a secure mobile group messaging application that supports offline clients, but has unclear guarantees regarding a total order of messages, and does not mention real-time constraints. The lack of a server also suggests that Mobile CoWPI cannot provide stronger-than-Causal Consistency, while SCUBA can.

Conclusion

SCUBA enables strong consistency for end-to-end encrypted applications under Byzantine actors. SCUBA proposes new mechanisms for achieving and validating strong consistency, and lets clients provably blame a misbehaving server for consistency violations. TANK, a key-value store with tunable consistency, adds permissions handling and invariant checking. We build four previously-unsupported applications on SCUBA, show that SCUBA imposes low overheads, and show that the SCUBA server can scale to realistic workloads. Ultimately, SCUBA safely brings strong consistency to many end-to-end encrypted applications.

Acknowledgments

We thank our reviewers for their invaluable feedback. This work was supported by the National Science Foundation under Grant No. 2443589, a Stellar Development Foundation Academic Research Grant, and a Google Faculty Research Award. Natalie Popescu was supported by a Microsoft Research PhD Fellowship, Shai Caspin is supported by an NSF GRFP No. 2039656 and a Gordon Wu Fellowship in Engineering, and Leon Schuermann was supported by a Francis Robbins Upton Fellowship in Engineering.

References

[1] Code of Federal Regulations, Part 164-Security and Privacy. https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164.

[2] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX annual technical conference (USENIX ATC 16)*, 2016.

[3] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.

[4] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrgelis, Bryan Ford, Jean-Pierre Hubaux, and Joan Feigenbaum. PriFi: Low-Latency Anonymity for Organizational Networks. *arXiv e-prints*, page arXiv:1710.10237, October 2017.

[5] Khashayar Barooti, Daniel Collins, Simone Colombo, Lois Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. Cryptology ePrint Archive, Paper 2023/880, 2023. https://eprint.iacr.org/2023/880.

[6] Mathias Biilmann. *Modern Web Development on the JAMstack*. O'Reilly Media, Incorporated, 2019.

[7] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[8] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1639–1656, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Debajyoti Das, Easwar Vivek Mangipudi, and Aniket Kate. Organ: Organizational anonymity with low latency. Cryptology ePrint Archive, Paper 2022/488, 2022. https://eprint.iacr.org/2022/488.

[10] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association.

[11] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[12] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[13] Malene Kulild Fredheim. Secure messaging with crypho. Master's thesis, NTNU, 2020.

[14] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 47–60, 2012.

[15] David Gobaud. messages out of order. https://github.com/signalapp/Signal-Desktop/issues/2424, 2018.

[16] Yuncong Hu, Sam Kumar, and Raluca Ada Popa. Ghostor: Toward a secure {Data-Sharing} system from decentralized trust. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 851–877, 2020.

[17] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

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

[19] Aaron MacSween, Caleb James Delisle, Paul Libbrecht, and Yann Flory. Private document editing with some trust. In *Proceedings of the ACM Symposium on Document Engineering 2018*, DocEng '18, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[21] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *11th USENIX Security Symposium (USENIX Security 02)*, San Francisco, CA, August 2002. USENIX Association.

[22] Moxie Marlinspike. Private group messaging. https://signal.org/blog/private-groups/, May 2014.

[23] Matrix.org. End-to-end encryption implementation guide, Sep 2022.

[24] Meteor0id. Messages not shown in the right order. https://github.com/signalapp/Signal-Desktop/issues/6156, 2022.

[25] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf, 2016.

[26] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 85–100, New York, NY, USA, 2011. Association for Computing Machinery.

[27] Redis Contributors. Redis pub/sub. https://redis.io/docs/manual/pubsub/.

[28] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. Solid: a platform for decentralized social applications based on linked data. *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.*, 2016.

[29] Michael Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society (WPES)*, WPES '19, pages 55–73, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Michael Schliep, Eugene Vasserman, and Nicholas Hopper. Consistent synchronous group off-the-record messaging with sym-gotr. *Proceedings on Privacy Enhancing Technologies*, 2018:181–202, 2018.

[31] Firebase Team. Firebase. https://firebase.google.com/, 2023.

[32] Health Team. Health Privacy Overview. https://www.apple.com/privacy/docs/Health_Privacy_White_Paper_May_2023.pdf, 2023.

[33] Keybase Team. Keybase. https://keybase.io/, 2023.

[34] Matrix Team. Matrix. https://matrix.org/, 2023.

[35] PreVeil Team. PreVeil. https://www.preveil.com/, 2023.

[36] ProtonMail Team. ProtonMail. https://proton.me/, 2023.

[37] Signal Team. Signal. https://signal.org/, 2023.

[38] Threema Team. Threema. https://threema.ch/en, 2023.

[39] Tresorit Team. Tresorit. https://tresorit.com/, 2023.

[40] Virtru Team. Virtru. https://www.virtru.com/, 2023.

[41] WhatsApp Team. WhatsApp. https://www.whatsapp.com/, 2023.

[42] Wire Team. Wire. https://wire.com/en/, 2023.

[43] Telegram FZ LLC and Telegram Messenger Inc. Telegram.

[44] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 172–182, New York, NY, USA, 1995. Association for Computing Machinery.

[45] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 423–440, New York, NY, USA, 2017. Association for Computing Machinery.

[46] u/abandoned faces. Missing messages. https://www.reddit.com/r/whatsapp/comments/11pey4n/missing_messages/, 2023.

[47] u/aezren. Missing messages on ios. https://www.reddit.com/r/Threema/comments/11eeqj9/missing_messages_on_ios/, 2023.

[48] u/BlindWolf8. Some messages missing on mobile only? https://www.reddit.com/r/Telegram/comments/oei13w/some_messages_missing_on_mobile_only/, 2021.

[49] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. In *IEEE Symposium on Security and Privacy*, 2015.

[50] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.

[51] Web Hypertext Application Technology Working Group (WHATWG). Html living standard. https://html.spec.whatwg.org/commit-snapshots/127962300e7d52f32273ee9f7a1a5ed9a6161e7a/, April 2023.

[52] Luke Williams. Messages send in incorrect order (attempting with no connectivity, then regaining connectivity). https://github.com/signalapp/Signal-iOS/issues/5241, 2022.