ChatGPT

Security Analysis of Dacument

Overview of Dacument and Its Security Features

Dacument is a client-side, schema-driven CRDT document framework with built-in cryptographic integrity controls. Every state change is packaged as a cryptographic token ("DACOP" type) and **digitally signed** by the actor's role-specific private key [1] [2]. This means each operation is tamper-evident and can be verified by other replicas using the stored public keys for the Owner, Manager, and Editor roles. Dacument enforces strict **role-based access control (RBAC)**: only certain roles can perform specific actions (e.g. only Owners/Managers can change ACL, only Editors and above can modify document fields) [3]. These role checks are applied at the time associated with each operation's timestamp (using a hybrid logical clock) to ensure historical ACL rules are respected [4] [2]. Importantly, Dacument is *"secretless"* on the server side – no private secrets or keys are stored in any centralized component; all sensitive keys reside with the clients (the participants) [5].

Some of Dacument's **notable security features** include:

- **Digital Signatures for Integrity & Authenticity:** Every mutation operation is signed with the actor's role key. Peers verify the signature against the corresponding public key (embedded in the document snapshot) before accepting an operation [2]. This prevents forgery or unauthorized edits – if a malicious party injects or alters an operation in transit, the signature check will fail and the op is rejected [6] [2]. Even the *initial document creation* and ACL assignments are signed by the Owner's key, establishing a chain of trust from the start [7] [8].

- **Role-Based Access Controls:** Each participant is assigned a role (Owner, Manager, Editor, Viewer, or Revoked) with specific permissions [3]. The library enforces that a signed operation is only applied if the actor's role (at the operation's timestamp) authorizes that action [3] [9]. For example, an Editor's key can only sign content changes (not ACL updates), and a Manager's key cannot create new Managers/Owners (only downgrade or grant Editor/Viewer) [10] [11]. Any operation that doesn't satisfy the role constraints (e.g. an Editor trying to perform an ACL change, or a Manager attempting to elevate someone to Manager role) will be rejected during merge/application [10] [11].

- **Strict Schema Validation:** The document schema defines expected data types and even regex patterns for fields. Incoming operations are validated against the schema – unknown fields or type mismatches cause the operation to be rejected [2]. This prevents malicious or accidental introduction of improper data. For instance, an attempt to assign an invalid value or a disallowed field is caught locally (or at merge on peers) and does not corrupt the state [12] [13]. This also mitigates certain injection risks, since only expected data types/values pass; an attacker can't, say, insert a script tag into a field that is defined as numeric or constrained by regex.

- **Op Log Integrity and Idempotency:** Dacument maintains a log of operations (op log) and uses a **hash/ID-based deduplication** to ignore duplicate or replayed operations. Each operation token is unique and stored in a set; if a token is seen again, it won't be applied twice [14] [15]. This means if an attacker tried to replay old operations, other peers would recognize the duplicate

token and ignore it (the op would be considered already applied) [16]. Combined with the signatures, this provides strong protection against replay or out-of-order tampering.

- **Acknowledgements (Ack) Mechanism:** For efficiency, Dacument uses unsigned "ack" operations to acknowledge receipt of updates and enable garbage collection of CRDT tombstones [17]. Security is maintained by special-casing acks: the library only accepts ack tokens if they use the `alg: "none"` (no signature) header *and* originate from a currently valid (non-revoked) actor [18] [19]. Any attempt to send a **signed ack** (which could be a malicious actor's attempt to fake another peer's acknowledgment) is explicitly rejected [17] [20]. Similarly, an ack claiming to be from an unknown or revoked actor is ignored [19]. This design prevents attackers from manipulating the garbage-collection acknowledgments (for example, to force premature data deletion or to impersonate other peers' acks).

- **No Persistent Secrets & Key Handling:** When a new document is created, Dacument generates a fresh public/private key pair for each privileged role (Owner, Manager, Editor) using secure randomness [21] [1]. It returns the private keys to the application and **does not store them internally** [5]. The onus is on the application to distribute these keys securely to the intended users. Public keys are included in the document snapshot so that any loader/replica can verify incoming ops against the correct keys [1]. Keys are never automatically rotated by Dacument [1], which simplifies the design but means key rotation (if ever needed) must be handled at the application level (e.g. by migrating to a new document with new keys). The absence of persistent secrets in the library means there's no secret vault to breach on a server – compromise of the server or transport won't yield document keys, since they were never there to begin with.

Potential Weak Points and Issues

While Dacument provides strong integrity and authorization controls, a security review reveals some **weak points and considerations** to keep in mind:

- **Shared Role Keys (Identity vs. Role Impersonation):** Dacument's model uses one key pair per role **per document**, not per user. This means all users assigned the same role share the same signing key for that document [1]. A significant implication is that the system does not cryptographically distinguish between different actors who have the same role. The only identifier in an operation is the actor's ID (`iss` field), which is not tied to the key via a unique signature – the signature only proves that *someone with the Editor key* (for example) signed the op, and the token's `kid` claims it's actor X [22] [9]. **This opens an impersonation risk among colluding or compromised same-role users:** a malicious Editor could forge operations that appear to come from another Editor's actor ID, and as long as they use the shared Editor private key and a plausible timestamp, other replicas will accept it as an op from the purported actor [9]. There is no per-actor public key to distinguish the true origin. The only defense is that actor IDs are large random nonces, so an attacker would need to know the target actor's ID (which in many cases is observable once that actor is added to the ACL). In practice, this means **non-repudiation is only at the role level**, not the individual level – all Editors are mutually trusted to act honestly, because if one turns malicious they can frame others or at least inject operations under any Editor identity. This is a design trade-off that simplifies key distribution but is a weak point for accountability. As a best practice, teams using Dacument should only give shared role keys to users who trust each other to that extent, or else consider operational mitigations (e.g. out-of-band auditing of clients or limiting the number of people per role).

- **Role Key Compromise Impact:** If a private key for a role is stolen or compromised, the attacker effectively gains all privileges of that role across the document. Because keys are not rotated and are shared per role, the compromise of one Editor's key is equivalent to compromising *all* Editors for that document – the attacker can sign any Editor operation and even impersonate any editor's actor ID as mentioned. If a Manager key is compromised, the attacker can perform any Editor-level changes and also modify the ACL for lower roles (they could revoke or downgrade other users, including **revoking the Owner** as discussed below) [10]. A stolen Manager key cannot promote the attacker to Owner (since Manager cannot assign Owner role by design [10]), but the attacker could cause significant disruption by removing the actual Owner's access rights (setting the Owner's role to "revoked") [10]. Indeed, the current implementation allows a Manager to revoke an Owner's access, effectively locking out the Owner; this could be seen as a security oversight or intentional design to let Managers remove a possibly compromised Owner. In any case, a Manager key compromise is nearly as severe as an Owner compromise in practice. If an Owner key is compromised, the attacker has full control – they can sign any operation including transferring ownership or granting themselves roles, and no other party can override those without the Owner key. **Mitigations:** At the first sign of key compromise, the recommended approach is to stop using that document (since keys can't be rotated) or immediately revoke the affected user roles in the document (which at least prevents further ops from that actor being accepted) [23]. However, revoking doesn't invalidate the stolen key outright – if other actors still use that same role key, the attacker could impersonate one of them. The safest resolution is often to start a fresh document and migrate the data, distributing new keys out-of-band.

- **Revocation and Out-of-Order Operations:** Dacument's handling of user revocation has security advantages but introduces a consistency wrinkle. When a user is set to "revoked," they immediately lose read/write privileges – their local doc will mask all data and refuse to generate ops [24]. Additionally, if other peers receive a revocation for actor X, the library will reject any further ops from X *even if those ops were sent earlier (timestamp-wise) but arrive later in real time*. In other words, once revocation is processed, any **out-of-order straggling ops** from that actor are invalidated and dropped on merge [23]. This is a security-minded choice: it prevents a revoked (possibly malicious) actor from sneaking in operations that were delayed in transit after they've been removed, which could be dangerous. The trade-off is that it can lead to temporary divergence if another peer had applied that straggling op before seeing the revocation. In practice, consistency can be restored by sharing a fresh snapshot from a peer with the "true" state. The important point is that **revocation is treated as higher priority than eventual consistency** for security reasons – once you declare an actor untrusted (revoked), the protocol favors disregarding anything from them that wasn't already integrated. This behavior is intentional and noted in tests (revoked actors' late ops are not applied) [23]. Application developers should be aware that revoking a user quickly halts their contributions and might lead to some data from them being lost if it wasn't delivered in time – essentially part of the threat model where availability is sacrificed to contain a potentially malicious actor.

- **No Built-in Encryption (Confidentiality):** Dacument's focus is on integrity and consistency; **it does not encrypt document data by itself**. All operations and snapshots are transmitted/stored as JSON (or JWT-like tokens) that contain plaintext field values, relying on the environment to provide confidentiality if needed [25]. This means that if you send Dacument ops or snapshots over an insecure channel (or store them in plaintext on a server), an eavesdropper or intermediary could read the document contents. The library is "secretless" in that it doesn't handle secrets centrally, but it's also *content-agnostic* with respect to encryption – the assumption is that you will use an authenticated, encrypted channel (more on that below) or encrypt at the application layer. Failing to do so is a security risk: while an attacker can't *modify* data without

detection (thanks to signatures), they could **passively surveil** all document changes if the transport is not encrypted. Also, an attacker who can man-in-the-middle the connection (without breaking encryption) still cannot forge ops due to signatures, but they could perform traffic analysis or replay old encrypted blobs if the application protocol doesn't guard against it. **Best practice:** Always run Dacument in an environment with strong transport security (e.g. E2E encryption via TLS, WebRTC Data Channels, or application-level encryption of each message). The documentation itself hints that Dacument doesn't provide transport – it's the developer's responsibility to wire up networking [25] – which implies the developer must also ensure that transport is secure (it's **not safe to rely on Dacument's signatures alone for confidentiality**).

- **Client-Side Trust and Exposure:** As a purely client-side library, Dacument's security assumes that the client application is not compromised. If an attacker can run arbitrary code in the user's browser or app (through XSS, malicious plugin, or other malware), they could intercept keys and messages, defeating all the cryptographic guarantees. For example, in a browser scenario, an XSS vulnerability in the hosting application could allow an attacker to call `doc.snapshot()` or read the in-memory state and keys, sending them out to a server. Similarly, since private keys are managed by the app, any weakness in how the app stores them (e.g. in localStorage without encryption, or logging them accidentally) would be a weak point. This is not a flaw in Dacument per se, but a reality of its threat model – the **endpoint must be secure**. Thus, standard web or app security practices (CSP headers, avoiding XSS, secure storage APIs, etc.) are crucial. Another exposure is that snapshots contain the entire history of the document's ops (including content) by design [26]. If a snapshot file is saved to disk or shared, treat it like sensitive data – anyone with the snapshot (and the schema to interpret it) can read the document's contents and see the public keys. They won't have private keys to forge new ops, but the existing content is visible. Encrypting snapshots at rest or in transit is advisable in sensitive applications.

- **Denial of Service Vectors:** As with many real-time collaborative tools, a malicious participant could attempt to flood the system with operations to degrade performance or bloat storage. For instance, an Editor could insert a huge volume of text or a large number of set entries in a short time. Dacument will faithfully sign and broadcast these, and other clients will attempt to merge them. The CRDT algorithm ensures the system won't *break*, but it could become slow or memory-heavy (CRDTs retain tombstones/metadata until GC, so excessive writes without acks can grow state) [17]. Another potential DoS angle is crafting operations that are technically valid but heavy to process (e.g., an Editor could send a patch with a very large number of inserts in one operation). The library does impose some checks (like the structure of patch nodes, data types, etc., which if not met will reject the op early) [27] [28], so purely invalid data won't propagate. However, a valid but extreme operation can still tax the system. Applications might need to implement usage limits or rate limiting at the app layer (since Dacument itself doesn't limit op rate or size beyond what JSON can encode). Additionally, because every signature verification uses CPU (ECDSA P-256 verification), an attacker who floods another client with a huge number of bogus signed messages (even if they can't be applied, the signature still has to be checked before rejection) could cause high CPU usage. In an E2E encrypted context, though, an outsider can't flood directly (they'd have to be an insider with a valid key to get their messages through decryption and into the merge function). So this is mostly the case of an already-authenticated user turning hostile.

- **Lack of Audit Trail per User:** Building on the shared-keys issue, auditing changes on a per-user basis is hard. The system emits merge events with an `{ actor, ... }` field indicating which actor ID performed a change [29], and one could log these. But if non-unique keys are used per role, those logs are not cryptographic proof of which human did it – just which actor ID was used. If two editors dispute who made a certain change, the cryptography can only prove "an

Editor did this," not specifically *which* editor (since all editor-signed ops look valid for any editor). This might be acceptable for small, trusted groups but is a weakness for larger or zero-trust collaborations. One mitigation could be having each user run their own Dacument document and merging at application level, but that forfeits real-time collaboration benefits. It's a known limitation given the current design.

In summary, Dacument's primary weaknesses are not in its cryptography implementation (which is solid) but in certain design decisions: **key sharing per role, no encryption, and reliance on client app security**. These issues mean that while an external attacker is very well-defended against, insider threats or misuse need to be managed by careful operational practices.

Secure Usage Context and Environment

Dacument is intended to be used in a **fully client-side, decentralized fashion**, which influences how to secure it. Key points about the recommended environment and usage include:

- **Client-Side Execution:** Dacument runs in modern browsers or Node.js (v18+ with WebCrypto) as an ES Module [30]. This means the security of the runtime (browser or Node process) is crucial. In a browser, it should be served over HTTPS and ideally with subresource integrity or other measures to ensure the script isn't tampered with in transit. On Node, avoid exposing the process to untrusted inputs beyond the Dacument messages themselves. Since no server component of Dacument holds state, any server is merely a relay; securing the server (to ensure availability and that it isn't hijacked to drop or delay messages) is still important but the server cannot decrypt or forge data if E2E encryption is used.

- **Networking – E2E Encryption and Authentication:** The typical deployment scenario for Dacument would have peers communicating over an **authenticated, end-to-end encrypted channel** – for example, a WebRTC DataChannel between clients that have verified each other, or an overlay network where messages are individually encrypted. Authentication ensures you know which peer you are talking to (preventing impostors), and encryption ensures confidentiality. When the user says the peer connection is authenticated and E2E encrypted, it implies that only the intended participants can read the data and inject messages. In such a scenario, Dacument's requirements are met: it does not itself encrypt, but it relies on the transport being secure [25]. If you were to use a simple server relay (e.g. sending ops via WebSocket to a server and out to others), you should at least use TLS (which is point-to-point encryption). However, note that a server in the middle with TLS can still see the plaintext of ops and snapshots. So if the server is not fully trusted, a better design is to perform an application-layer E2E encryption of the ops. This could be done by encrypting the `event.ops` payloads before sending, using a group symmetric key or pairwise encryption between peers. The "secretless" nature of Dacument means *it doesn't help you perform this encryption*, but it also doesn't interfere – you are free to encrypt the blob of ops and have the receiving side decrypt before calling `doc.merge(ops)`.

- **Suitable Use Cases:** Dacument is best used in applications where a small group of known clients collaborate on shared state (e.g. a collaborative text editor, shared to-do list, or CRDT-based data sync in a P2P app) and where **trust is distributed among clients rather than a server**. It shines in scenarios where you don't want to trust a server with your data or enforcement logic. Each client keeps a local copy of state and only exchanges signed deltas. This fits peer-to-peer or "local-first" software models. For instance, if building a secure collaborative note-taking app, you'd generate a Dacument per note, have one user as Owner (who can invite others by giving

them keys and snapshots), and exchange updates via a secure channel. In contrast, Dacument is *less appropriate* if you wanted a big public collaborative space with untrusted strangers – the key sharing model and lack of per-user audit would be problematic, and managing keys for thousands of users would be cumbersome. It's also not meant for server-side enforcement: since any client with the Owner key can grant roles, you would not centrally moderate beyond what the app itself allows.

- **Key Distribution Environment:** Because keys are generated on creation and then must be shared securely, think about how your application will deliver keys to the right people. For example, when a new Editor is to be added, an existing Manager or Owner must out-of-band send the Editor private key to that person (perhaps via an encrypted invite link or by scanning a QR code in person). This should be done over a secure channel (never email or plaintext messaging). The environment should treat role keys as secrets akin to passwords or private certificates. If the app is web-based, you might use the Web Share API or a messaging system with end-to-end encryption to share keys. **Never embed role private keys in client code or URLs in an unprotected way** – if an attacker finds an invite link that contains a key, they can join as that role.

- **Actor IDs and Identity:** Dacument requires you to set a unique actor ID (256-bit random) per device/process using `Dacument.setActorId()` [31]. The environment should ensure a truly random ID is used (the provided `generateNonce()` does this with cryptographic randomness). Collisions are astronomically unlikely with 256-bit IDs, but using a proper random source is essential. The actor ID has no inherent personally identifiable info – it's just an internal identifier. However, in some environments you might link it to a user identity in your app logic (e.g. mapping Dacument actor IDs to usernames in your UI for display). Be careful that any such mapping is for user convenience only; the security model inside Dacument only knows about actor IDs and their roles, not human-readable identities.

- **Storage Considerations:** If your app saves Dacument snapshots or op logs (for offline or backup), remember these contain all document data. Store them in secure storage (browser IndexedDB, filesystem, etc.) with encryption at rest if the device is not fully trusted. Mobile apps might use the OS keystore to encrypt files. The good news is that if an attacker obtains a saved snapshot without any role keys, they can read the data but cannot produce new valid changes. If they also get a role private key from storage, then it's game over for that document's security. So protect those keys wherever they are stored (consider using secure enclaves or prompting the user for a passphrase to decrypt a stored key, depending on threat model).

- **Compatibility and Updates:** Keep the environment up to date. Dacument leverages WebCrypto; ensure the runtime's crypto libraries are patched and not using deprecated algorithms. ES256 (ECDSA on P-256) is currently secure, but implementation bugs in crypto libraries can occur – using the latest Node and browser versions helps mitigate known issues. Also, monitor the Dacument repository for any security patches. At the time of analysis, no specific security issues in the code were publicly noted (tests cover many edge cases and invalid input handling looks robust [6] [32]). But staying on the latest version will ensure you have any improvements or fixes.

Best Practices for Maximum Security

To maximize security when using Dacument, consider the following best practices and mitigations:

- **Use End-to-End Encryption for Transport:** Always send Dacument ops (`event.ops`) and snapshots over an encrypted channel. Ideally, establish peer-to-peer encryption that the server cannot intercept (for example, using WebRTC or an encryption library). This ensures that only authorized clients can see the document's content. It also means that if someone intercepts the network traffic, they gain nothing (the ops are gibberish to them). Remember that while Dacument's signatures prevent undetected tampering, they do not hide content – encryption is needed for privacy.

- **Authenticate Peers and Validate Identities:** Ensure that you're really talking to the right peer. If using WebRTC, use signaling with authentication (e.g., exchange identity keys or certificates). If using a server relay, have a login or identity verification such that you don't accept ops from unknown sources. One approach could be to include the expected actor IDs or public keys in an initial handshake, so if an unexpected actor ID shows up, you flag it. Dacument's own verification will reject ops signed with wrong keys, but a fake user could flood messages (which get rejected but still consume resources). Authentication protocols at the session level help filter out such noise early.

- **Limit Role Key Sharing and Scope:** Limit the number of people who get high-privilege keys. For example, perhaps only one Owner and one backup Owner exist. Use Manager roles sparingly if at all, and only for those who truly need to delegate. The fewer people holding a given key, the lower the chance of compromise or misuse. In contexts where you have many collaborators, you might consider segmenting the document (e.g., multiple documents each with a smaller set of editors) rather than one giant shared edit pool. This way, a compromised key only impacts a subset of data. **Never reuse role keys across documents** – always use the fresh keys generated by `create()` for each document to avoid cross-document impact.

- **Secure Key Distribution and Storage:** When you give someone a role key, do it through a secure method. If your app is web-based, you might send an invitation link that contains an encrypted blob of the key that only the recipient can decrypt (for instance, using their login credentials or a pre-shared secret). Encourage users to keep their keys private – for instance, if keys are stored client-side, don't expose them in debug logs or UI. If the application is multi-device (a user wants to use the document from phone and laptop), provide a safe means for them to transfer their key to the second device (QR code scan, etc., rather than emailing it). Remind users that **losing control of a private key is equivalent to losing control of the document's security for that role**.

- **Implement Revocation Procedures:** If a user's device is lost or you suspect a key is compromised, use the ACL to revoke that actor immediately. Dacument will propagate the revocation as an op, and from that point on, that actor (even if an attacker has their key) cannot make changes [23]. For example, if an Editor's laptop is stolen, an Owner or Manager should revoke that Editor's ID in the doc. This doesn't invalidate the key itself cryptographically, but it updates the ACL so that any op stamped after the revocation time from that actor is ignored. In tandem, generate a new Editor key for the user (perhaps by removing and re-adding them as a new actor with a fresh ID/key, if continued access is needed). It's a good practice to have a procedure or UI in your app for such emergency removals.

- **Monitor and Log Operations:** Keep an eye on what operations are coming in. Since each merge event tells you which actor (ID) performed an operation and what it was, consider logging this (with timestamps) to an audit log that the team can review. While, as discussed, this isn't foolproof for attribution if role keys are shared, it can still help detect abnormal behavior. For instance, if you see a flood of edits from an actor who normally is quiet, that might signal a compromise or a bug. Or if an Editor suddenly tries to do an ACL change (which would be rejected, but you can catch the attempt in an `error` event or by the fact that merge result had a rejection), that's a red flag. Dacument emits an `"error"` event for signature/verification issues [29] – log those too, as they could indicate someone tried to inject a bad op. Essentially, treat the client as if it were a server in terms of logging security-relevant events (just don't log sensitive data unintentionally).

- **Sanitize and Handle Document Content Safely:** If the document content might be rendered as HTML or used in code, continue to sanitize on output. Dacument ensures the content is what valid users entered, but a malicious Editor could still enter a string like `"<script>alert(1)</script>"` as a field value if the field is a free-form text and not disallowed by regex. If your app later injects that string into a webpage without escaping, it could trigger XSS in whoever views it. So maintain normal content security hygiene: escape user-provided text when displaying, validate any URLs or references, etc. In other words, Dacument doesn't magically make user content safe – it only ensures the content came from authorized users. You still should not trust that content beyond the schema constraints. Use the schema's regex and type features to limit content where possible (as was done in the example for the title field allowing only letters and spaces [33] [34]). This reduces the burden on sanitization for those fields.

- **Plan for Key Rotation (If Needed):** Although Dacument doesn't support rotating keys within a document (the snapshot's public keys are effectively permanent for that doc [1]), plan operationally how you would handle a long-lived document. For example, if a document is expected to live for years with different participants, you might periodically create a new document and migrate the data (especially if membership changes significantly). This achieves a crude form of key rotation by introducing new keys with the new document. All existing data can be snapshotted and loaded into the new doc (since an Owner can always create a snapshot and presumably start a new doc with that state, giving out new keys). This is admittedly heavy-handed, but it ensures that a document doesn't have the same keys forever (which could be risky as more people have had access over time). In short, *treat each document's lifetime as linked to the trust in its keys* – if trust changes, consider starting fresh.

- **Testing and Simulation:** If you're building a security-critical application on Dacument, test the threat scenarios. Simulate a malicious collaborator by writing unit tests or using the library's own test techniques: e.g., try to sign an operation with a wrong key or an outdated timestamp and confirm it's rejected (the library's test suite does this already [6]). Simulate a dropped message and late arrival after revocation to see how your app UX handles it (one peer might show an edit that another peer never applies – maybe you'll show a warning or force a resync). By anticipating these edge cases, you can build user-facing handling (perhaps informing users "User X's changes were not applied due to revocation" or similar). Leverage Dacument's events (`error`, `revoked`) to provide feedback: for instance, if the current user gets revoked, the library emits a `revoked` event so you can alert them or restrict the UI [29].

In summary, achieving maximum security with Dacument requires combining its strong **built-in guarantees** (integrity, authenticity, ACL enforcement [2]) with external measures for **confidentiality and key management**. Use encrypted channels, carefully manage who holds keys, and respond swiftly to any sign of compromise. When used in the right context (small, trusted collaborations), and with

these practices, Dacument can provide a very robust security posture: an attacker who is not an authorized participant should find it nearly impossible to read or modify the data, and even authorized users are constrained by the role-based rules.

Threat Model in an Authenticated E2E-Encrypted Setting

With the assumption that the peer connections are **authenticated and end-to-end encrypted**, we can outline the threat model – i.e., what attacks are still possible and which are mitigated – for a Dacument-based application:

- **Outside Eavesdropper:** A network attacker who can sniff traffic (e.g., someone on the same Wi-Fi, or a malicious ISP) **cannot read document data**. The E2E encryption of the channel means they only see ciphertext. Since each operation is encrypted, the attacker also cannot replay or modify operations in a meaningful way – without the encryption keys, they can't even craft a valid ciphertext that the peers would accept. Thus, classic confidentiality threats (sniffing, passive surveillance) are addressed by the transport encryption. Additionally, because operations are signed, even if this attacker somehow could inject a message, it would fail verification at the recipient and be rejected [6] [2]. So both confidentiality and integrity are covered for outsiders.

- **Malicious Server or Relay:** If your architecture uses a server to relay messages but the content is E2E encrypted (the server cannot decrypt it), then the server is effectively in the same position as an outside attacker. It could drop or delay messages (denial of service), but it cannot alter them undetectably, nor read them. Dacument's CRDT nature tolerates delays and reordering, so delays just affect sync speed, not final correctness (unless the server permanently drops some messages – which would be a DoS requiring user intervention to resolve, such as re-sending a snapshot). The server could try replaying old messages out of context, but thanks to signatures and the op log, replicas will ignore duplicates or already-applied ops [14]. A malicious server could attempt a **replay of a prior state** (sending an old snapshot to a new client, for example), but if the client also receives real-time updates from honest peers, it will eventually converge to correct state (and the outdated snapshot's ops would either be superseded or found invalid if they conflict with signed history). To fully guard against a malicious relay, clients might include sequence numbers or use an agreed-upon hash of the latest state to detect if something's missing – though this is more on the application protocol level. In summary, a malicious server can degrade service but not violate confidentiality or integrity if E2E encryption is used and Dacument's verification is in place.

- **Unauthorized Participant (No Key):** Consider an attacker who does not have a valid role key but somehow tries to participate – perhaps by guessing the docId and connecting to the peer network, or by obtaining an invite link without the key. Because the transport is authenticated, they shouldn't even join the session unless they pretend to be someone else. But even if they manage to inject messages, they cannot produce a valid signed operation without a private key. All their attempts will fail signature verification and be rejected [6]. They also can't decrypt others' messages if they lack the E2E encryption key. So an attacker without keys is effectively locked out on all fronts. The threat model here is very strong: such an attacker's best bet would be denial of service (flooding the network with garbage data or trying to knock clients offline), but not data compromise. Authenticated channels usually have mechanisms to drop or ignore messages from unknown senders, further reducing this risk.

- **Compromised or Malicious Participant:** This is the **primary remaining threat** in an E2E scenario. If an attacker manages to become an authorized participant (e.g., they convince an

Owner to give them an Editor key, or they steal someone's key/device), they now operate from *inside* the trusted boundary. Dacument's design will limit what they can do based on their role, but within those bounds they can certainly cause mischief. For example, a malicious Editor can add incorrect or malicious data to the document (which is a valid action – the system can't know the difference between a legitimate edit and a harmful one like flooding the text with spam or inserting offensive content). They cannot escalate privileges on their own (an Editor's signed ACL change will be ignored) [10] [11], which is good – they can't just make themselves Owner. A Manager with malicious intent is more dangerous: they could revoke other users or downgrade them, perhaps as a form of sabotage or to monopolize control. However, a Manager still cannot make themselves an Owner or add new Managers, so the damage is somewhat contained to harassment and denial (revoking others) unless an Owner intervenes. A malicious Owner is effectively the worst-case insider – they can do anything (including distributing new keys to bad actors, deleting or altering data, etc.) [3]. The **threat model must assume that owners are fully trusted**; if an Owner account is taken over by an attacker, the document's integrity and confidentiality are broken (the attacker could even strip away encryption if they control the app, or just leak all the content).

Mitigations within this threat category include the social and procedural: only give out keys to those you trust, monitor the document for strange actions, and use revocation quickly if someone goes rogue. It's worth noting that the cryptography won't stop a *malicious but authorized* user from simply copying sensitive information out of the document and sharing it elsewhere – once they can read it, they can leak it. End-to-end encryption and signatures don't prevent this form of data exfiltration by an insider. So sensitive deployments might use legal agreements or additional monitoring (for example, watermarking content or detecting large copy-paste events) to discourage insider leaks. At a technical level, one could limit how much data a single op can contain (preventing an editor from, say, pasting 10,000 lines in one operation as a way to flood). Dacument doesn't enforce such a limit, but an application could impose one before calling `doc.merge` or before sending ops out.

- **Man-in-the-Middle with Stolen Keys:** If an attacker steals a participant's key and then attempts to impersonate them on the network, what happens depends on the scenario. In an authenticated E2E channel, typically each participant has a known identity (like a device identity or TLS certificate). An attacker with just the Dacument role key would also need to subvert the network identity to truly MiTM or impersonate that user in real time. If they can do that (e.g., steal Alice's key and also her login or her device connection), then effectively they are a malicious participant as above. Dacument will treat them as Alice (if they use Alice's actor ID and key) and other clients will accept ops from them, thinking Alice performed them. This is why protecting private keys is paramount. However, note that if the real Alice is still online and the attacker tries to use the same actor ID, there might be unusual effects (two sources generating ops claiming to be the same actor). The system doesn't have a built-in notion of distinct sessions – it would merge ops from both, and if the ops conflict or interleave strangely, the CRDT will still order them by timestamp. But operationally, this could confuse the real Alice (seeing changes she didn't make). This is more of an edge-case, as normally one actor ID = one device/client. The threat model could include **device or account takeover** – if that happens, the device is effectively an attacker. Standard device security (OS-level protections, 2FA for app login, etc.) come into play here; Dacument can't solve it. The recourse is again revocation: as soon as the compromise is noticed, revoke that actor so the stolen key can no longer be used to affect the document [23].

- **Side-Channel Analysis:** In a fully encrypted system, an attacker might resort to side-channels like traffic analysis. They might observe message sizes or timing to infer something (e.g., typing

bursts vs. idle, or that a large snapshot was sent indicating a big change). If this level of adversary is a concern (perhaps in very high-security contexts), mitigations could include adding padding to messages or sending constant-size heartbeats to mask activity patterns. This goes beyond Dacument's scope, but it's part of a comprehensive threat model if facing sophisticated opponents (like state-level actors).

- **Collusion and Trust Boundaries:** If multiple participants collude maliciously, they could attempt to subvert processes (for instance, a Manager and an Editor colluding: the Manager grants the Editor a higher role temporarily to do something and tries to cover it up). However, every ACL change is an auditable op and can't be erased (unless all replicas agreed to wipe history). So collusion would be evident in the op log (e.g., "Manager X granted Editor Y Owner role at time T and later changed it back"). Other participants would see those ops unless they were completely offline and only got a snapshot after the fact – but snapshots include the op log, so they'd still see it; though a snapshot might be pruned of tombstones, it doesn't remove valid ops. Thus, Dacument provides transparency that makes silent privilege escalation difficult to hide. The threat model assumes all non-revoked actors' ops eventually reach all others [17] [2], so misbehavior is eventually visible. Collusion mainly could be used to bypass application-level rules (like if the app UI prevents Managers from doing something but they script it via Dacument directly – but that's an app design consideration).

In summary, **with authenticated, encrypted channels, the threat landscape is largely reduced to insider threats** and device compromise. Dacument very effectively neutralizes external attackers – they can't read data, can't inject fake changes, and can't replay things to gain an advantage. The remaining risks come from those who have legitimate access: either abusing their allowed powers or having their credentials stolen. The best defenses there are organizational (careful assignment of roles, quick revocations, user training) and application-level (monitoring, perhaps limits on usage patterns, secure coding to protect keys). It's also important to note that Dacument's approach to security is optimistic: it assumes participants want consistency and will eventually share all ops. If an insider simply withholds ops (e.g., refuses to ever acknowledge, to prevent tombstone GC and bloat others' storage), the system can't force them – it just won't garbage-collect in that case [17]. That's a possible denial tactic (a rogue who stays offline to prevent GC), albeit a minor one unless storage is a concern.

Overall, the threat model for Dacument in a properly secured environment is quite strong against external threats and standard network attacks. The focus shifts to *trust management*: ensuring the right people hold the right keys and handling the scenario when a trusted party goes bad. When used as intended (small groups of mutually trusted or at least accountable users), Dacument provides a high level of security for collaborative state management. All participants can verify that every change was authorized and untampered (via signatures) [2], and thanks to encryption, they can be confident no one outside the group is reading their data. The design achieves a balance where no central server needs to be trusted with secrets or writes – the clients collectively enforce the rules. Users of Dacument should remain aware of the few but important weak points (shared role keys and client security) and mitigate those with the strategies discussed. With that in place, one can confidently use Dacument for end-to-end secure collaboration, knowing its cryptographic guardrails will hold up under all but the most dire insider scenarios.

Sources:

- Dacument README – *Dacument: schema-driven CRDT with signed ops and role-based ACL* [35] [1] [36]
- Dacument Source Code (GitHub: `jortsupetterson/dacument`) – *Implementation of signing, verification, ACL enforcement, etc.* [9] [10]

- Dacument Test Suite – *Demonstrations of security behaviors (rejection of invalid ops, unsigned acks, role enforcement, etc.)* 20 6

---

1 2 3 4 5 17 25 26 29 30 31 33 35 36 README.md
https://github.com/jortsupetterson/dacument/blob/fba8ddbbfdab95a1bdfd9d067ab4abc7446a292f/README.md

6 11 12 13 20 23 24 32 34 dacument.test.js
https://github.com/jortsupetterson/dacument/blob/fba8ddbbfdab95a1bdfd9d067ab4abc7446a292f/test/dacument.test.js

7 8 9 10 14 15 16 18 19 21 22 27 28 class.ts
https://github.com/jortsupetterson/dacument/blob/fba8ddbbfdab95a1bdfd9d067ab4abc7446a292f/src/Dacument/class.ts