ChatGPT

Tools and Libraries for a C2PA-Enabled React Native Video Capture App

Introduction

Building a **React Native** app that captures live video and embeds **C2PA-compliant** provenance metadata involves combining several components: a robust video capture module, a library/SDK to generate and embed C2PA manifests (signed metadata), cryptographic tools for key management and signing, and a workflow for certificate provisioning. Below, we outline the most efficient tools and libraries for each aspect (for both **iOS and Android**), and compare key options and trade-offs. We also discuss how to manage certificate authority (CA) integration and signing workflows on-device vs. via an AWS backend.

Video Capture in React Native

Capturing high-quality live video in a React Native app can be achieved with well-established libraries. These libraries provide a cross-platform Camera module to record videos (and often capture photos) with consistent APIs:

Library	Platforms	Features	Considerations
React Native Vision Camera (open-source)	iOS, Android	Modern, high-performance camera library with GPU-accelerated pipeline. Supports video recording (H.264/H.265 codecs) with configurable quality, frame rates, flash/torch, etc [1]. Also supports real-time frame processing and AR use-cases.	Actively maintained (community-driven). Requires native setup (CameraX on Android, AVFoundation on iOS). Well-suited for capturing video and getting the output file path in RN.
Expo Camera (Expo AV) (open-source)	iOS, Android	Part of Expo's API (or unimodules). Simplified camera access for recording video and photos. Easy to use in managed Expo apps.	Easiest if using Expo, but less customizable than Vision Camera. If using bare React Native (no Expo), Expo Camera can still be installed but adds overhead.
React Native Camera (deprecated)	iOS, Android	Legacy camera library (supports video capture, photos, basic processing).	No longer actively maintained – superseded by Vision Camera. Not recommended for new projects due to outdated dependencies.

Library	Platforms	Features	Considerations
FFmpegKit (react-native-ffmpeg) (open-source)	iOS, Android	Not a camera per se, but a powerful video processing toolkit. Can be used after capture for tasks like trimming, compression, or adding data via FFmpeg commands [2].	Use for post-processing (requires including FFmpeg binaries). Not needed just for capture, but useful if complex video manipulation is required beyond what camera libs offer. May increase app size and complexity.

Recommendation: For most use cases, **React Native Vision Camera** is the optimal choice for capturing live video. It provides efficient, configurable recording and is designed for React Native's architecture. VisionCamera's API gives you the video file path or URI upon finishing recording [3], which can then be used for post-processing (embedding metadata). Expo Camera is an alternative if your app is built with Expo and you prefer not to eject – but if fine-grained control or performance is needed, Vision Camera is superior. FFmpeg-based libraries can supplement for heavy video editing tasks (if needed), but embedding C2PA metadata can typically be done without full re-encoding.

On-Device Video Post-Processing

After recording, the app needs to **embed a C2PA manifest** into the video file (or as a sidecar) – this is a form of post-processing. There are two general approaches:

- **Using a C2PA SDK/Library:** The preferred method is to use a dedicated C2PA library that can attach a signed manifest to the video file in-place. The open-source C2PA tools (detailed in the next section) handle this by inserting the manifest (which includes cryptographic signatures and assertions) into the media file's metadata sections (e.g. into an MP4 container or image's metadata) [4] [5]. This avoids having to manually manipulate bytes or re-encode the video. The C2PA library will produce an output file that is identical to the input video except for the added tamper-evident metadata.

- **Using a Media Processing Library (e.g. FFmpeg):** In theory, one could use a generic video processing library to insert metadata (for example, injecting an XMP or binary metadata box into the MP4). However, doing this in a C2PA-compliant way is non-trivial without the actual C2PA toolchain, because it's not just embedding text but also generating cryptographic signatures and following the specific standard. FFmpeg or similar could be used to **augment** the video (e.g. transcode, resize), but for **C2PA manifest insertion and signing, a dedicated C2PA library is necessary**.

In summary, for C2PA metadata embedding, it is most efficient to leverage an existing **C2PA SDK** that handles manifest creation and insertion, rather than attempt to script this via generic video processing. We cover these SDK options next.

C2PA-Compliant Metadata Embedding SDKs

The **Coalition for Content Provenance and Authenticity (C2PA)** provides an open standard and reference implementations for embedding signed provenance metadata ("Content Credentials") into media files. A number of SDKs and libraries have emerged to support C2PA:

SDK / Library	Platform / Language	Capabilities	Integration in RN	Trade-offs
Content Authenticity Initiative (CAI) Open Source SDK – C2PA Rust Core + native wrappers (Adobe & partners)	Core in Rust (c2pa-rs); Wrappers in C++, Swift, Kotlin	Full-featured reference implementation of C2PA. Supports reading, validating, and **writing signed manifests** into supported media formats (JPEG, PNG, MP4, etc.) [6] [5]. Mobile-specific SDKs available: **c2pa-ios** (Swift) and **c2pa-android** (Kotlin) wrap the Rust core for iOS/Android use [7] [8].	Requires creating a React Native native module to bridge to Swift/ Kotlin. The iOS SDK is distributed via Swift Package (XCFramework) [9]; Android via AAR (Maven) [10]. You would call its functions (e.g. `signFile`) through native modules.	**Pros:** Official, spec-compliant implementation; maintained by C2PA contributors (Adobe, etc.) so likely up-to-date with spec changes. Pre-built binaries and example apps make integration easier [11] [12]. Supports flexible signing options (e.g. custom assertions, and *callback signers* to use external keys) [13] [14]. **Cons:** Currently in **alpha** (APIs subject to change) [15]; documentation and community are growing but not as large as general RN community. Will require native bridging (no out-of-the-box RN module yet).

SDK / Library	Platform / Language	Capabilities	Integration in RN	Trade-offs
Guardian Project “Simple C2PA” (Open-source)	Cross-platform (Rust core, with Kotlin and Swift bindings via UniFFI)	A simplified mobile-friendly C2PA library built on the same Rust core. Provides high-level methods to add signed assertions to media files on-device [4]. Supports generating keys and even self-signed certificates locally for quick start [4] [16]. Focuses on ease of use for camera apps (originally integrated in the ProofMode app).	Can be added as a dependency: available via Maven (`info.guardianproject:simple-c2pa`) for Android [17] and Swift Package for iOS [18]. Similar to above, would need RN bridge to call functions (e.g. `addContentCredentials`). In practice, this library influenced the official c2pa-android/ios SDKs, and the codebases may be converging.	**Pros:** Mobile-optimized and relatively simple API. Allows starting with self-signed certs (no immediate backend needed) [16]. Proven to work with JPEG and MP4 (the main formats of interest) [5]. **Cons:** Early-stage (as of late 2023 it was “stable alpha” [19]). May not have full feature parity with the official SDK (though largely similar). If the official contentauth SDKs are now available (as of 2025) and stable, those might supersede this.

SDK / Library	Platform / Language	Capabilities	Integration in RN	Trade-offs
Truepic Lens SDK (Commercial)	iOS, Android (closed-source SDK)	A commercial offering providing a **secure camera** and C2PA signing. Truepic's SDK (called *Truepic Lens*) allows apps to capture images (and possibly videos) with built-in content credentials. It produces C2PA-compliant photos and is backed by Truepic's certificate authority and secure processing [20].	Integrating requires obtaining the SDK from Truepic. It's a drop-in module that handles capture and signing internally. For RN, you might need a native module wrapper unless Truepic provides RN support.	**Pros:** Turnkey solution – handles camera capture, signing, and verification in one. Truepic is a leader in image authenticity, and their system ensures all content is signed with a trusted certificate (after a product conformance process). **Cons:** Not open-source; requires partnership or license. Less flexibility (tied to Truepic's workflows and cloud backend). Primarily focused on photos (C2PA for video may or may not be supported in their SDK – needs confirmation).

SDK / Library	Platform / Language	Capabilities	Integration in RN	Trade-offs
Numbers Protocol Capture SDK (Commercial)	iOS, Android (closed-source SDK)	Another commercial SDK (by Numbers Protocol's *Capture*) enabling on-device C2PA signing. It is advertised for enterprise use to sign user-generated content (including images and video) on mobile, with features like automatic daily key rotation and blockchain integration for provenance tracking [21] [22].	Similar integration considerations as Truepic – would involve adding their native SDK and possibly bridging to RN. Likely provides a high-level API to capture or to submit media for signing.	**Pros:** Enterprise-ready, comes from an early C2PA contributor [23]. Could simplify compliance by handling the intricacies of signing and certificate management (they mention Truepic certificates and secure key handling [24]). **Cons:** Commercial and closed. Brings in additional complexity (e.g. their blockchain features) that might be beyond the needs of a custom solution. Also, reliance on their backend/ services.

SDK / Library	Platform / Language	Capabilities	Integration in RN	Trade-offs
DIY Integration (Custom Native Module)	iOS, Android (custom C++/Obj-C/Java)	Roll your own integration by using the low-level C2PA libraries (Rust/C++). For example, compile the core C2PA Rust library (or use the C++ API bindings [25]) for mobile and invoke it via the React Native C++ bridge or JSI.	This approach means writing a native module that calls C2PA functions (e.g. using JNI for Android and a Swift/Obj-C wrapper on iOS). You might do this if you want full control or need to customize the library.	**Pros:** Maximum flexibility and no third-party runtime dependencies beyond the C2PA core. You can stay very close to the spec and update the library as needed. **Cons:** Heavy lift in development effort – essentially re-implementing what the official mobile SDK wrappers already do. Only consider if the existing wrappers don't meet requirements (e.g. if you need a custom build of the C2PA library with specific patches).

Recommendation: Leverage the **open-source C2PA SDK** provided by the Content Authenticity Initiative (the Rust-based implementation with iOS/Android wrappers) for on-device metadata signing. By 2025, these official mobile SDKs have matured, offering comprehensive features for manifest creation and verification [26] [27]. Integrating them into React Native would involve writing minimal bridging code: for instance, exposing a function like `embedProvenance(videoFilePath, metadata, cert, key)` that internally calls the native SDK's `signFile` or equivalent [28] [29]. This provides a trusted, spec-compliant way to embed C2PA manifests.

The **Guardian Project's SimpleC2PA** is a viable alternative if you prefer its simplified API or if it provides a feature you need (e.g. on-device key generation) not directly in the official SDK. Notably, the official Android

SDK supports *callback-based signing*, allowing integration of custom signing logic (which can be used to interface with secure hardware or external services) [13] [14]. This flexibility means you can still use hardware-kept keys (via callbacks) with the official library, making it very adaptable.

Commercial solutions like **Truepic** or **Capture SDK** could be considered if time-to-market and certification are paramount – they effectively outsource the heavy lifting (and come with their own CA trust anchors). However, given you operate your own CA and have a custom AWS backend, using the open SDK and integrating with your backend gives more control. It avoids vendor lock-in and aligns with building an open, **custom pipeline (Terraform-managed AWS backend + your CA)**.

Cryptographic Libraries for Key Storage & Signing

C2PA relies on **digital signatures** and X.509 certificates, so your app needs a way to **manage cryptographic keys** securely and perform signing operations according to C2PA specs (e.g. typically ECDSA or RSA signatures on the manifest). Key considerations include protecting the private key (ideally in secure hardware), being able to generate signatures on messages (manifests), and verifying signatures as needed.

In a React Native context, you have a few options for cryptography:

- **Use Platform Key Stores via Native Modules:** Both iOS and Android provide secure key storage natively. On iOS, the **Keychain** (with Secure Enclave) can generate and store keys that never leave the enclave. On Android, the **Android Keystore** (with hardware-backed modules like StrongBox or TEE) does similarly. To interface with these in React Native, you can use libraries or native modules:
- **React Native Device Crypto** (open-source) is a library that provides a high-level API to generate and use keys in secure hardware from JavaScript [30] [31]. It supports creating an asymmetric key pair (private key stays in Secure Enclave/KeyStore) and then using it to sign data [32] [31]. It also integrates optional biometric authentication flows (so that signing can be gated by user presence if desired). This is ideal for our case: the app can generate a key on-device, and use `sign()` to sign the manifest bytes with the private key, all without exposing the key in memory.
- **React Native Biometrics** (open-source) is another library with similar capabilities geared towards authentication. It can create keys in the secure hardware and sign payloads (often used for login challenges, but it's general purpose).

- **React Native Keychain** (open-source) is more about storing secrets/passwords in Keychain/ Keystore. It's useful if you have to store an existing private key securely (e.g. if the key is generated elsewhere and imported, though typically keys should be generated in the secure store to avoid ever existing in plaintext). Keychain could store a PEM or PKCS12 securely, but using native crypto APIs directly is safer than extracting keys.

- **Use the C2PA SDK's Cryptography (with PEM keys):** The C2PA libraries themselves include cryptographic routines (often leveraging OpenSSL or Rust's *ring* library under the hood) and can accept keys in software form (e.g. a PEM-encoded private key string). For example, the iOS SDK's `SignerInfo` can be created from a PEM-encoded private key and certificate [28]. However, *storing* the key as PEM in the app's storage is risky; for production, this is not recommended without strong encryption. The documentation explicitly warns not to leave private keys accessible on the file system and instead use a secure key store [33]. In practice, this means that if you use the SDK in a straightforward way, you should load the key from a secure source (not a plaintext file). One approach is to combine this with the above: retrieve the key from Secure Enclave using RN Device Crypto (e.g. export the public key for certificate, but keep private key hidden) and then use a *callback signer* so the C2PA SDK asks your code to sign the digest. The Android SDK already supports such callbacks [13], and the iOS SDK can likely be extended or modified to do similar, if not already supported. This way you **don't pass the raw key to the library**, only the signature.

- **Web3/Crypto Libraries in JavaScript:** There are some JS libraries (like `tweetnacl`, `jsrsasign`, or Node's `crypto` polyfills via packages like `react-native-quick-crypto`) that can perform cryptographic operations. However, handling X.509 and C2PA's required algorithms purely in JS is complex and unnecessary given the native options. Using the OS facilities or the C2PA's built-in crypto will be more secure and efficient. For example, the C2PA spec requires support for **ECDSA (ES256, ES384)** and **RSA-PSS** signature algorithms [34], which are fully supported by the SDK (and by device crypto APIs). If you tried to do this in JS, you'd still need to manage private keys securely, which leads back to using native storage anyway. Thus, it's best to leverage native/hardware crypto for key storage and signing.

Recommendation: Generate and store the signing key in secure hardware on-device, and use native crypto APIs (via a library like *react-native-device-crypto*) to perform signatures. This ensures the private key is protected by the Secure Enclave/TEE – the key cannot be extracted, and signing can require user biometric confirmation if appropriate. The C2PA SDK can be configured to use this key: on Android, use the callback signer interface to supply signatures without exposing the key [14]. On iOS, if a direct callback interface isn't available yet, one strategy is to have the RN bridge retrieve the data to sign (the C2PA SDK's C layer can be tweaked to use an external signer) – or use the approach of signing a sidecar and merging it. In any case, **avoid keeping private keys in plaintext** on the device. The use of a Key Management Service or hardware module is advised in production [33], which in our mobile context translates to the OS secure key store. This aligns with best practices: *"Accessing a private key directly from file system is fine during dev, but in production is not secure. Instead use a KMS or HSM…"* [33].

For **verification** of signatures (if the app needs to verify content credentials), the C2PA SDK will handle that as well when reading a file. The SDK will validate the signature against the certificate and even check against a trust list of known CAs if provided. If you needed to manually verify on the backend, you could use standard cryptographic libraries (OpenSSL, BoringSSL, or AWS KMS to verify signatures with the public key). But on-device, simply calling the SDK's verify/read functions will suffice – they will return whether the manifest is valid and details of the signer.

Certificate Provisioning and Signing Workflow

Managing **certificates** is a crucial part of a C2PA system. Since your organization operates as a **registered certificate authority**, you have the ability to issue the **Claim Signing Certificates** that your app will use. Here we outline how to provision certificates and design the signing workflow, both on-device and via the AWS backend:

- **Provisioning Device Certificates:** Each app instance (or each user/device) will need an **end-entity certificate** that is used to sign content. This certificate's private key should correspond to the secure key on the device. A typical workflow:
- **Device-Generated Key + CSR:** When the app is first installed or on first launch, it generates a new key pair in its secure enclave/keystore. It then creates a Certificate Signing Request (CSR) containing

the public key and device identity info (this step may require a small native helper or a JS library that can create a CSR from a key; OpenSSL on a server or a Node module can also create the CSR if the public key is sent).

- **CA Issuance:** The CSR is sent to the backend (over a secure channel). Your AWS backend, managed via Terraform, could be configured with a **Private CA** (for example, AWS Certificate Manager Private CA) or a custom CA service. The backend verifies the request (you may enforce authentication to ensure only legitimate app instances get certs) and uses the CA to issue an X.509 certificate. This certificate will chain up to your organization's CA root (or intermediate). The certificate should include the extended key usage and other fields required by C2PA (the C2PA spec defines certain X.509 extensions for provenance signing certs [35] – ensure your CA template includes those).
- **Certificate Delivery:** The backend returns the newly issued certificate (and the rest of the chain) to the device. The app stores the certificate (this is public info, not secret) and likely also the intermediate CA cert, so it can embed the chain in the C2PA manifest. The C2PA SDK expects you to provide the certificate (and any chain) along with the signing operation so that the manifest includes the correct certificate chain [36]. Including the chain ensures that verifiers can trace the signature to a trusted root (either your root, if it's in their trust list, or a known root if you become part of the C2PA Trust List). *Note:* If your CA is part of the official C2PA Trust List, validators (like Adobe's Verify tool) will recognize it; otherwise, you may need to distribute your root cert to your verification clients or use a private trust store [36].

- **Renewal/Revocation:** Certificates might have an expiration (e.g. 1 year or even shorter if you prefer). The app should handle renewing the certificate by repeating the process as needed (perhaps transparently when near expiry). If a device is compromised or a key is suspected to be leaked, your CA backend should revoke that certificate (via a CRL or OCSP mechanism). Currently, C2PA trust validation can consider certificate revocation status if configured. Since devices may work offline, revocation checking might not always occur immediately, but you can push an updated trust list or require re-validation when online. The C2PA spec notes that participants should promptly revoke compromised certificates [37].

- **On-Device Signing:** With a provisioned certificate and a secure private key, the device can sign content **entirely offline**. This is aligned with the C2PA design goal of supporting offline capture devices [38]. For example, a journalist's camera app could capture a video in the field with no connectivity, and still produce a C2PA-signed video. The certificate needed was obtained beforehand, and the signing uses the locally stored key. C2PA confirms this scenario: *"Devices like cameras can securely generate and sign Content Credentials using locally stored cryptographic keys, without needing internet. Certificates can be provisioned in advance or renewed later when connectivity resumes."* [38]. In our case, the mobile app acts as that offline device when needed. The manifest can include the signing timestamp (the SDK can also contact a timestamp authority if available, but offline it will use the device clock). Once back online, the app or others can verify the content using the certificate chain.

- **Backend Signing (alternative workflow):** In some architectures, you might choose to perform signing on the backend instead of on the device. For instance, the device could upload the captured video to AWS, and the backend (which holds the signing key in a **Hardware Security Module** or AWS KMS) attaches the C2PA manifest server-side. AWS published a guidance example using this approach: a Lambda/Fargate service takes an asset and produces a C2PA sidecar file, with the private key stored in AWS Secrets Manager and used by the C2PA CLI in a container [39] [40]. The sidecar (or signed asset) is then returned to the client. You could implement a similar pipeline with

Terraform provisioning the necessary AWS resources (e.g. an ECS task running the open-source `c2patool` or the Node.js SDK to sign assets, with keys in AWS KMS or Secrets Manager). The benefit of backend signing is that the private keys never reside on user devices – they can be kept in a centralized HSM with strict access control [41]. This significantly limits attack surface for key compromise. It also allows using high-assurance keys (possibly hardware-backed by AWS CloudHSM or KMS with multi-party controls).

- **Trade-offs – On-device vs. Backend:** There are trade-offs between these two signing models:

- *On-Device Signing:* Low latency (sign immediately as part of capture workflow), works offline, and the user's device holds their own signing identity. This empowers user autonomy and matches scenarios like secure camera hardware. However, it means distributing trust – each device has a key that could be a point of attack if the device is compromised. Mitigation comes from using hardware key storage and the ability of the CA to revoke certificates if needed. Also, if your org's root is widely trusted, each device's signatures will be recognized; if not, it might be an internal trust model.
- *Backend Signing:* Centralized control of keys (easier to protect in one place). You can use powerful HSMs and tightly monitor signing. It simplifies revocation (you could simply stop signing for a device if it's not trusted). The downside is reliance on connectivity and added round-trip for each capture. Users must upload potentially large videos to get them signed, which could be slow or impractical in the field. It also introduces an architectural dependency – the capture process is no longer self-contained in the app, which might hurt user experience or robustness in patchy network conditions.

In many cases, a **hybrid approach** can be employed: for example, the device does on-device signing for basic provenance (so that something is attached even if offline), but later the backend can re-sign or countersign the asset when it's uploaded (embedding an additional assertion or replacing the provisional signature with a higher-assurance one). C2PA's format supports multiple assertions and even multiple signatures (ingredients and updated manifests), so an asset could initially have a self-signed or local credential, and later an official timestamp or cloud-verified credential can be added. This, however, adds complexity and is an advanced scenario.

Given that your organization is a CA, you likely will issue **trusted certificates to the devices** and go with on-device signing for real-time capture. Ensure that your certificate issuance process and Terraform scripts cover: - **AWS Private CA** (or an equivalent) for managing your CA hierarchy. Terraform can automate the setup of an AWS PCA, issuance policies, and even create an interface (API Gateway + Lambda) for signing CSRs. - **AWS KMS** or Secrets Manager for storing the root CA's private key (if using AWS PCA, AWS manages the key in HSM for you). This keeps the root key secure. Intermediate CAs can also be managed similarly. Terraform can provision these and output the necessary info (like the CA ARN). - **Device Enrollment API:** A secure endpoint for the app to request a certificate. This could verify the app (e.g. require the user to authenticate, or use device authentication tokens). Once approved, the backend uses the CA to issue the cert. You might integrate this with your user management so each cert is tied to a user account or device ID. - **Certificate Revocation Lists (CRL) or OCSP:** If using AWS PCA, it can maintain a CRL in S3. Make sure your verification process (and possibly the C2PA SDK's trust store) is aware of revocation. In an internal ecosystem, you might periodically push an updated trust list to clients, excluding any revoked certs (the C2PA verify tool uses a trust list of known valid signers [42]).

Managing Trust: Because C2PA is aimed at a broad ecosystem, ideally your CA certificate (or a root you chain to) is in the **C2PA Trust List** – a list of CAs that are considered valid for content credentials [35] [43].

Achieving this requires going through the C2PA Conformance Program (as noted in the FAQ, only products that undergo evaluation get certificates from trust-list CAs [44]). If your goal is broad interoperability (e.g. any C2PA consumer will trust your signatures), you might pursue that route. If it's a closed ecosystem (only your apps or partners need to trust the content), you can maintain your own trust anchors. In either case, when embedding the manifest, include the full certificate chain up to (but not including) the root. The C2PA SDKs will take your `certificatePEM` chain and embed it; validators will then attempt to chain to a root they trust [45] [36].

Comparison of Key Options and Trade-offs

To summarize the recommendations in a comparative view:

- **Video Capture:** *React Native Vision Camera* is the top choice for capturing video in-app, offering performance and flexibility. Expo Camera is suitable for simpler needs or managed workflow. Ensure you handle permissions and test on both iOS and Android for any device-specific quirks (like focus, stabilization, etc. which VisionCamera supports via config).

- **C2PA Metadata Embedding:** The *open-source C2PA SDK (Rust-based with mobile wrappers)* provides a proven path to add provenance info on-device. It supports all necessary formats (including MP4 for video) and is kept up-to-date with the standard [5]. It will handle manifest generation, cryptographic hashing of the media, and embedding the signature and metadata into the file. Using this library via a React Native bridge is efficient: the heavy work is done in native code (Rust/C), so performance overhead is minimal – important when dealing with large video files. Alternative open tools like SimpleC2PA can be used if needed, but they serve a similar function. Commercial SDKs (Truepic, Numbers Protocol) are available if compliance and fast integration outweigh the desire for a fully custom solution; however, given you have internal CA capabilities, the open SDK gives you flexibility to integrate with that.

- **Cryptography & Key Management:** Use **device secure storage** for keys (e.g. Secure Enclave, Android Keystore). Libraries like RN Device Crypto abstract a lot of this and make it straightforward to generate keys and sign data with minimal code, all while keeping keys protected [31]. Align the key algorithm with C2PA spec (ECDSA with P-256 (ES256) is a common choice for C2PA credentials, providing a good balance of security and performance [34]). Verify that the chosen algorithm is supported by both the device hardware (most devices support P-256 ECDSA in hardware) and the C2PA SDK (ES256 is supported by C2PA toolkit). Storing keys in hardware also means if the app is uninstalled, keys are destroyed, and a new cert would be needed – this can be a feature (each install = new identity) or something to mitigate (backup the key in secure cloud if you need continuity – though that reintroduces key exposure, so usually not done).

- **Certificate Authority Integration:** Having your own CA gives maximum control. Use Terraform to manage the AWS resources for CA and possibly an issuance microservice. Keep the root key in a secure AWS KMS or HSM. Possibly use an **Intermediate CA** that issues the device certificates, so the root can be kept offline or in a very limited HSM. The intermediate's cert can be in the trust list. Automate certificate issuing and renewal so that it's seamless (e.g. device requests could include an auth token, and the backend signs a cert valid for, say, 3-6 months to limit risk). This way, even if a device's key is somehow compromised, its cert will eventually expire and cannot be misused indefinitely.

- **On-Device vs Cloud Signing:** Favor on-device signing for user-generated content at capture time, to meet the real-time and offline requirement. Backend signing can be reserved for additional processing (for instance, if later the video is edited in cloud or needs a **secondary signature** by a server confirming something). Always ensure that at least one signature (the device's) is attached at creation so the provenance is recorded early. The backend can always validate the manifest it receives (since it will trust its own CA) and could add a *verified* assertion if desired.

Below is a brief **trade-off table** for signing approaches:

Signing Approach	Pros	Cons	Use Case
On-Device (Device-held key & cert)	– Immediate signing at capture (no network needed).<br/>– Works offline; user retains control of their credential. [38] <br/>– Keys can be device-specific, reducing blast radius of a key compromise.<br/>– Leverages device hardware security for key protection.	– Must securely provision certs to many devices (PKI overhead).<br/>– Revocation of a compromised device cert might be slow to propagate if offline.<br/>– Device integrity is critical (if device is jailbroken, key could be misused until revoked).	Best for **edge capture scenarios** (e.g. cameras, mobile in field). Ensures content is signed at moment of capture with minimal delay. Use when trust in device and user is acceptable and connectivity is uncertain.
Backend (Server-held key & signing)	– Centralized key custody (easier to secure one key in HSM). [41] <br/>– Allows high-assurance signing (server can enforce checks, add timestamps, etc).<br/>– Simplifies client app (no PKI on device, just upload data).<br/>– Revocation is straightforward (if device misbehaves, just refuse to sign).	– Requires network uplink for each signing; higher latency and no offline capability.<br/>– Scalability considerations for large media (upload/download overhead).<br/>– Single point of failure/target (if server or key is compromised, affects all content).	Best for **controlled environments** or cloud workflows – e.g. signing content as it enters a system (media library, AI generated content on server). Useful if devices are thin clients or untrusted. Not ideal for live mobile capture due to connectivity dependence.

Signing Approach	Pros	Cons	Use Case
Hybrid (Device signs, Server re-signs or attests)	– Combines advantages: initial offline credential from device, followed by an authoritative signature from server when possible.<br/>– Server can attach additional **attestations** (e.g. "Verified by server at upload time").<br/>– If server signature is present, verifiers can choose to trust that more if they trust server's CA.	– Implementation complexity (need to manage multiple manifests/updates in one asset).<br/>– Larger file due to multiple credentials (though still only KBs).<br/>– Requires careful design to avoid confusion between signatures.	Use for **multi-stage workflows**. E.g., a news camera app signs locally, then newsroom server adds its own signature confirming the footage was received unaltered. Ensures redundancy in trust.

Finally, ensure that all components are working together: the **React Native app** will capture video, call the **C2PA SDK via a native module** to embed the manifest, using a **certificate** issued by your **AWS-managed CA** and a **private key** stored in secure enclave. The result is a video file (MP4) that contains a cryptographic provenance trail. When this file is uploaded or shared, any compliant viewer or verification tool can inspect the C2PA manifest to see the signer (your organization/device), the time, and any other assertions you include (location, editing software, etc., as per your needs). Because you control the CA, you can decide the level of identity tied to the certificate (it could be a per-device anonymous ID, or it could embed user info or device IDs in the cert's subject or in the C2PA assertions).

Conclusion

By combining **React Native Vision Camera** for capture, the **Content Authenticity Initiative's C2PA SDK** for metadata signing, and strong cryptographic practices (secure enclave key storage and a robust PKI backend), you can build a mobile app that captures video and immediately seals it with verifiable provenance. The use of Terraform to manage AWS resources like a Private CA and key management services will help automate and secure the server side of this workflow. This stack ensures that the videos produced are **C2PA-compliant**, cryptographically signed at the source, and traceable to your organization's certificate authority – providing a chain of trust from capture to verification. Users and downstream platforms will be able to verify that the content is authentic and see the embedded origin metadata, fulfilling the goals of content provenance and authenticity.

Sources: The recommendations above are informed by the C2PA open-source documentation and SDK features [7] [8], insights from the Guardian Project's mobile C2PA work [4] [16], and best-practice guidelines for key management in C2PA [33] [38].

---

[1] [3] Recording Videos | VisionCamera
https://react-native-vision-camera.com/docs/guides/recording-videos

[2] ffmpeg-kit-react-native - NPM
https://www.npmjs.com/package/ffmpeg-kit-react-native

[4] [5] [16] [17] [18] [19] Simple C2PA for Mobile: Content Credentials To Go! – Proofmode
https://proofmode.org/blog/simple-c2pa

[6] [25] C2PA C++ library | Open-source tools for content authenticity and provenance
https://opensource.contentauthenticity.org/docs/c2pa-c/

[7] [9] [11] [15] [26] [28] GitHub - contentauth/c2pa-ios: C2PA SDK for iOS
https://github.com/contentauth/c2pa-ios

[8] [10] [12] [13] [14] [27] [29] GitHub - contentauth/c2pa-android: C2PA SDK for Android
https://github.com/contentauth/c2pa-android

[20] Truepic's New SDK Will Power Trusted Photo Capture Across the Internet
https://www.truepic.com/blog/truepics-new-sdk-will-power-trusted-photo-capture-across-the-internet

[21] [22] [23] [24] Capture C2PA Signing | Provenance for Google, Meta, TikTok, and OpenAI
https://captureapp.xyz/products/c2pa-signing

[30] [31] [32] GitHub - arifaydogmus/react-native-device-crypto: Cryptographic operations inside the secure hardware for React Native
https://github.com/arifaydogmus/react-native-device-crypto

[33] [36] [45] Signing with local credentials | Open-source tools for content authenticity and provenance
https://opensource.contentauthenticity.org/docs/signing/local-signing/

[34] JavaScript library | Open-source tools for content authenticity and provenance
https://opensource.contentauthenticity.org/docs/js-sdk/getting-started/overview/

[35] [38] [43] [44] C2PA FAQ
https://c2pa.org/faq/

[37] C2PA Security Considerations
https://c2pa.org/specifications/specifications/2.0/security/Security_Considerations.html

[39] [40] Guidance for Media Provenance with C2PA on AWS
https://aws.amazon.com/solutions/guidance/media-provenance-with-c2pa-on-aws/

[41] Signing and certificates | Open-source tools for content authenticity and provenance
https://opensource.contentauthenticity.org/docs/signing/

[42] Content Credentials : C2PA Technical Specification
https://c2pa.org/specifications/specifications/2.2/specs/C2PA_Specification.html