Certificate Pinning Done Right

Why Current Practice Pins the Wrong Thing

Glenn L. Austin, AustinSoft.com

Glenn Austin is an independent mobile security architect operating through AustinSoft in Mount Juliet, Tennessee. He has approximately fifty years of computing experience, including mobile security architecture at Atlassian and earlier work at Apple, Adobe, and Symantec. He holds US Patent 10,972,253 for cryptographic systems. The research presented in this paper was conducted independently as part of AustinSoft's consulting practice; the author has no affiliation with or financial interest in any certificate authority.

tl;dr

Standard certificate pinning implementations pin to the SHA256 hash of a certificate's public key — an unstable cryptographic identifier that changes whenever the issuer rotates keys, regardless of whether the organizational identity being authenticated has changed. This creates two failures of the security mechanism it was meant to provide.

First, current pinning misses an undetected attack vector: because root-level pinning verifies only that a chain terminates at the pinned root — not the specific intermediates and leaf along the way — a proxy operator who obtains a legitimate certificate from the same root CA can intercept pinned traffic undetected.

Second, with leaf certificate lifetimes being reduced to 47 days by 2029 (CA/Browser Forum, as reported by DigiCert, May 16, 2025), pinning to the leaf's public key requires coordinated app updates on the same cadence — operationally untenable for mobile applications going through MDM deployment cycles.

This paper presents the approach pinning should have taken from the beginning: pin to the identity expressed through the certificate chain's naming conventions that certificate authorities must adhere to under CA/Browser Forum Baseline Requirements to maintain trusted root status. Names enforced by root program governance are a stronger commitment than cryptographic key material rotated at CA operational discretion.

Practical implementation requires four string matching primitives. The security improvement is significant. The operational burden is lower than what you're doing now.

The Security Gap

You've been tasked with implementing communication security to guarantee against MITM attacks against your mobile application's infrastructure. Doing your security due diligence, you settle on certificate pinning to make sure that the server you ask for is actually the one you requested. The OS-level pinning APIs in iOS and Android handle the direct path, but you know that leaf certificates rotate frequently — and with certificate lifetimes shrinking, pinning to a leaf would mean shipping an app update every time a certificate renews. You research the options and decide to pin to just the root certificate of your CA provider, which allows the leaf and intermediate certificates to rotate freely beneath a stable anchor. One pin, no per-rotation updates, and every legitimate certificate your servers present still validates. You deploy your application and consider the MITM threat addressed.

Then the security calls start to come in. Someone has exploited your MITM defense — inserting malicious payload into the app, modifying its behavior to manipulate users into destructive choices, and even exfiltrating session credentials. In the worst cases, the attacker has pivoted from the compromised clients into your servers. The attacker never broke your pinning; they simply obtained their own certificate from the same public CA whose root you pinned, and your application accepted it as readily as it accepted your own. The root you pinned to allow your certificates to rotate authorized every other certificate that CA had ever issued — including the attacker's.

So, what went wrong?

Note that this isn't a failure of certificate pinning as a concept; it's a failure of how the industry has implemented it. The dominant approach uses the *root* certificate's Subject Public Key Information (SPKI) as the identifier — verifying an artifact of how the certificate was generated rather than verifying what the certificate represents. This choice is functionally equivalent to installing a Schlage deadbolt on your front door and only finding out later that *any* Schlage key will open the lock.

First, and most importantly, there is the problem of SPKI, which involves verifying the *identity* of the entity that created the certificate chain, not the *actual certificates* themselves.

Second, the SPKI matching mechanism that legitimately addresses leaf-level rotation gets propagated to root-level configurations through the assumption that certificates should be handled uniformly regardless of chain position. This propagation produces the master-key dynamic — the pinned root authorizes every certificate the root has issued *and will ever issue*, regardless of whether the application's threat model intended that authorization scope.

Third, 'common knowledge' as applied to security needs to come with explicit trade-off analysis for each decision it recommends. Pinning to the root CA only — the trade-off between operational simplicity and the master-key validation it produces. Using SPKI as the pinning identifier — the trade-off between cryptographic specificity and verifying an artifact rather than an identity.

Each of these decisions has defensible reasoning behind it; none of them comes with the trade-off articulation that would let the engineer evaluate whether the recommended approach fits their specific threat model and operational context.

Let's look at each of these problems separately.

Verifying the Identity vs. Verifying the Object

Subject Public Key Information, or SPKI, is using the X.509 certificate field containing the certificate's public key. Pinning libraries hash this field to produce a fixed-length identifier that can be compared against expected values configured by the application. Because each certificate in a chain is signed by the issuer's private key, SPKI hashing identifies the *signing entity* rather than the *signed certificate*. These private keys can be compromised, and have been on many occasions. The historical record demonstrates that this is not theoretical. CA-issuer credential compromise has occurred through multiple distinct mechanisms across more than a decade: direct infrastructure compromise (DigiNotar, 2011)[1], business-model delegation failures (Comodo, 2011)[2], operational misconfiguration (TURKTRUST, 2013)[3], systematic compliance failure (Symantec, 2017-2018)[4], and recent compliance-driven root program action (Entrust, 2024-2025)[5] for the issuer's failure to conform to the rules governing a Certificate Authority. Each

demonstrates that the issuer identity that SPKI pinning validates is not stable in the ways the pinning literature assumes.

Why was this a problem?

For *leaf* certificate validation, SPKI matching is actually well-suited. The leaf certificate has the most volatile amount of information-date of issue, date of expiration, what server(s) are covered by the certificate, what features the certificate enables, and what the common name is for this certificate-which is determined by the requester within constraints set by the CA's domain validation process. With the relative volatility of the leaf certificate in relation to other certificates in the trust chain, using SPKI to validate the leaf certificate makes perfect sense: the issuer can and *probably will* use the same intermediate certificate's signing key, which means that the SPKI hash will be unchanged.

This created a deeper problem in how libraries handled certificates. X.509 carefully distinguishes between root certificates (self-signed trust anchors), intermediate certificates (delegation points), and leaf certificates (the entity being authenticated). Each position has different semantic meaning and different validation requirements. But pinning libraries treated certificates as a homogeneous collection of objects to scan — implementing pin matching as a loop iterating through certificates checking for matches, with no awareness of position in the trust chain. The X.509 structural information that distinguishes root from leaf was discarded at the application validation layer. The "is member of set" composition that emerged from this flattening could check whether a pin matched somewhere in the chain, but couldn't enforce position-specific validation.

This is the Data Structures 101 instinct applied to a context where it fails. When you have a collection of same-typed objects, the trained response is uniform handling through iteration. Certificate pinning is the context where same type does not mean same role — and treating certificates uniformly discards the positional semantics that X.509 carefully encodes outside the type system.

All of this sets up the next part of the failure chain: the fallback mechanism that emerges from set-membership composition produces the master-key dynamic when applied to root-level configurations, because the libraries can't distinguish what they're matching against.

The Uniform-Handling Failure

The shift from full-certificate matching to SPKI matching limited the rotation problem at the leaf level because the key used was under the organization's control. With SPKI matching, the leaf's public key could generally remain stable — reissuing the certificate with the same keypair preserves the pinned key — so the pin could survive most renewals. It breaks only when the leaf's key itself is rotated, and key rotation is proper key management. Therefore SPKI and proper key management are in conflict: key rotation breaks SPKI, and SPKI discourages it. This conflict can only be resolved, when using SPKI, by pinning to a certificate known to have a long lifespan: the root certificate. The recommendation collapsed multiple distinct options into a single one: pin at the root, and use SPKI.

The consequence of this decision is that the default security posture is *insecure*, not secure.

By making the decision to pin to the root certificate, that pinned certificate becomes the reference for what trust chains are permitted. This is the master key problem noted above — any trust chain that terminates at the pinned root is considered legitimate, regardless of which intermediate signed the leaf or which entity the leaf was issued to. The pin authorizes everything the CA issues *and will ever* issue using the same root, which grows continuously as the CA continues operations. The application's exposure isn't bounded by what existed when the pin was configured; it expands with every certificate the CA subsequently issues — including any certificate an attacker legitimately obtains from that CA.

Root Rotation Disruption

When a root rotation event does occur, which is rare but does happen, the impact is substantial. CA infrastructure changes, algorithm transitions, key compromises, and compliance-driven distrust all force migration to new root infrastructure. Applications pinned to the affected root face a coordination problem with no good solution: certificates need to be renewed under the new hierarchy, applications need updated pins, users need to receive the updated applications before the compromised certificates expire. The mobile update tail makes this particularly painful — older application versions can't easily be updated, and users running them lose connectivity until they update or servers need to support those problematic certificates. Beyond the connectivity loss, the forced update or

migration itself sheds users — a known attrition cost, since some fraction of users abandon an app rather than update it, and a forced update (where the app stops working until updated) drives higher attrition than a voluntary one. The disruptive root event thus carries a direct business cost — lost users — not merely an operational one. Multiple organizations face this coordination simultaneously when major CA events occur. The operational simplicity that root pinning was supposed to provide turns out to defer the operational cost rather than eliminate it, with the deferred cost concentrated in a single, disruptive event rather than distributed across frequent small updates.

The same conflict between SPKI and proper key management noted for leaf certificates recurs at the root. Proper CA security practice requires generating new keys when rotating roots — limiting compromise blast radius, supporting key lifecycle management, enabling algorithm migration — and new keys mean new SPKI hashes. SPKI root pinning is therefore structurally incompatible with proper CA key management: either the CA reuses keys to preserve pin continuity, compromising its own security, or it follows proper practice and breaks every application pinned to the old root. The conventional wisdom recommending SPKI root pinning implicitly assumes one of these, and neither is acceptable.

The result is a deployed practice that compounds two independent mistakes. The primitive validates the wrong identity: SPKI pinning authenticates the signing authority rather than the connection, so the pin admits everything that authority has issued or will issue. The composition then widens rather than narrows that admission: pinning at the root — the broadest authority in the chain — applies the primitive at exactly the level where the wrong-identity problem is largest, and disjunctive matching across pin sets adds further acceptable chains rather than constraining to one. Neither choice was unreasonable in isolation. SPKI was a sound refinement over full-certificate hashing; root pinning was a rational response to the conflict between SPKI and key rotation. The failure is in their combination: a primitive that over-admits, applied at the level that maximizes the over-admission.

Both moves share a single pattern. A technical refinement, made with clear awareness of where it applied, became a general convention through propagation that dropped the awareness. SPKI hashing was scoped to the leaf, where it solved a

real problem; it propagated to the root, where the problem it solved doesn't exist and the problem it creates is worst. The original refiners weren't wrong within the scope they could see — they were solving the leaf-rotation problem, and they solved it. What propagated forward was the mechanism without the reasoning that bounded it. This is worth naming because it is not specific to pinning: any security primitive refined for one context and then adopted as a general default is exposed to the same drift, and the defense against it is to carry the original scope-reasoning forward with the mechanism rather than the mechanism alone.

Common Knowledge vs. Accuracy

Common knowledge in technical fields propagates faster than verification. A correction to a specific problem becomes generalized into a categorical principle, and the generalization carries the authority of the original correction even when applied beyond the original scope. Examples of this pattern are relevant to certificate pinning.

The first example was developed across the previous sections. Full-certificate hashing produced operational fragility at the leaf because legitimate certificate variation broke pins. The SPKI refinement was the scope-appropriate response: hash the public key rather than the full certificate, preserving the pin's validity across renewals. The refinement was bounded — leaf-level, addressing the leaf-level fragility problem. The propagation carried SPKI hashing beyond the leaf to root pinning through implementation symmetry rather than through analysis of whether the original scope-awareness still applied. The deployed convention became 'SPKI is the modern way to pin' without retaining the context that SPKI authenticates the signing entity rather than the certificate, which is the appropriate level of identity at the leaf but not at the root.

A specific cognitive substitution makes this propagation persist in practice. Practitioners deploying SPKI pinning at root or intermediate chain positions treat the validated chain as proxy for leaf certificate legitimacy. The reasoning is intuitive — the root signed an intermediate that signed the leaf, so a validated root chain produces a validated leaf. The substitution treats root validation as equivalent to leaf validation, even though the root SPKI pin only confirms that some certificate in the chain was signed by the pinned root's key. Operationally, both legitimate and adversarial chains validate identically when they share the

pinned root. There is no observable difference between "correct leaf signed by trusted CA" and "adversarial leaf signed by the same trusted CA." The system reports success in both cases, which makes the security property loss invisible to the practitioners deploying the configuration.

The pattern's depth is evident in platform vendor documentation. Apple's official guidance for ATS certificate pinning recommends storing the certificate authority's certificate as `ca.pem` for pinning, with the runtime configuration key `NSPinnedCAIdentities` encoding the assumption that Certificate Authority identities are the appropriate pinning target. The platform also provides `NSPinnedLeafIdentities` for leaf-level pinning, but the implementation recommendation doesn't discuss when this alternative would be appropriate or what trade-offs it addresses. Apple's recommendation reflects a defensible engineering judgment for the typical application case, where operational simplicity matters and the master-key authorization scope isn't a primary concern. The gap is that the guidance doesn't articulate the scope of applications the recommendation fits, which leaves engineers in security-sensitive contexts without the analysis they need to make informed decisions. Engineers following the standard pattern receive the rationale for pinning without the analysis of pinning-target trade-offs — finding the latter requires careful reading of the broader ATS documentation that most engineers configuring pinning won't do.

Google's Network Security Configuration documentation describes `pin-set` entries with backup pin support, framing the configuration in language that suggests primary pin plus fallback pin where either matching is sufficient. Engineers operationalizing this for production systems often configure a leaf pin plus a root pin together — the intent being to get leaf-level specificity normally with root-level fallback during certificate rotation events. The implementation, however, performs strict AND composition: all configured pins must match somewhere in the chain. During normal operation when the leaf matches the leaf pin and the root matches the root pin, both conditions are satisfied and validation succeeds. During leaf rotation events when the new leaf doesn't match the configured leaf pin, validation fails despite the root pin still matching the root — because the implementation requires all configured pins to match, not just one of them. The 'backup' configuration the engineer intended doesn't actually function as a backup; the engineer's mental model and the implementation behavior

diverge in ways that produce operational failures the engineer expected the configuration to prevent.

Engineers reading this critique reasonably want to know whether their own pinning configurations exhibit the problems documented here. The evaluation requires asking specific questions about what the configuration actually authorizes.

To evaluate whether existing certificate pinning provides the security property the configuration appears to specify, ask: at what chain position does the pin operate? If the answer is the root or intermediate position rather than the leaf, identify what other certificates the same issuer has signed. Public CAs sign certificates for millions of organizations. Private CAs — operated by an organization for its own application's certificates — sign only for that application. The pin's actual trust surface is the set of all certificates the pinned issuer has signed, which may be substantially wider than the configuration suggests.

Another piece of common knowledge is that the Common Name field has been deprecated across X.509. The deprecation is real but its scope is narrow: CN was deprecated for leaf certificate hostname matching, where Subject Alternative Name provides more capable hostname identification[6]. CN remains the standard identity field for issuer certificates throughout the trust chain, where its content is constrained by CA/Browser Forum Baseline Requirements[7] and root program inclusion policies. The content authority structure differs by chain position. At the issuer chain positions, CN content is effectively contractual — enforced by external governance through CA/Browser Forum requirements and root program policies. This provides stronger commitments than the internal key management practices that SPKI pinning depends on for stability. At the leaf, CN content is requester-controlled subject to CA validation, where requester-controlled content made the field unreliable for hostname identification. The common-knowledge propagation lost this distinction, dismissing CN categorically when the appropriate response was to scope the deprecation to the specific use case where the field's content authority structure made it unreliable.

Two intuitions made this outcome feel correct while it was happening. The first is that opacity signals security — that an inscrutable hash is a stronger guarantee than a name a human can read. The second is that sophistication signals quality —

that a cryptographic operation is a more serious engineering choice than a string comparison. Both intuitions are wrong here, and both pointed the same direction: toward the opaque, complex SPKI hash and away from the readable, simple name. The readable name is the stronger pin, and the simpler mechanism is the better engineering. Neither intuition survives contact with what the certificates actually do — which is why the question is ultimately empirical, and why the implementations that follow are tested against real certificate chains rather than argued from first principles.

All of the examples follow the same pattern. A legitimate technical concern produces an appropriate response. The response propagates as common knowledge. The propagation generalizes beyond the original scope, and the generalization persists because authoritative sources cite each other without continuous verification. Practitioners absorb the propagated version without re-examining whether the scope still applies to their specific context.

With these problems identified, two questions remain: does the failure mode appear in deployed code as the analysis predicts, and what alternative would address the structural causes? The first question is answered by examining current implementations directly. The second is answered by applying existing certificate validation mechanisms to the right identity fields, governed by the right authority structure, with composition semantics that produce the security property pinning was meant to provide.

Verification and Testing

The sources of the testing applications and tools used in this section are located at https://github.com/AustinSoftCom/Research-Better-Pin.

I built a test matrix of eight different pinning variants, using the current SPKI certificate pinning model against Apple's `captive.apple.com` endpoint:

1. All pins valid
2. Root pin valid, leaf pin invalid
3. Leaf pin valid, root pin invalid
4. All pins invalid
5. Root-only, valid

6. Leaf-only, valid
7. Root-only, invalid
8. Leaf-only, invalid

This allowed me to test the existing operating system implementations for both iOS and Android using the same pins and methods for both. The applications themselves read those entries and display them on the screen for validation during testing.

> A note on iOS methodology: iOS caches Info.plist contents based on the application's bundle identifier. Testing multiple pinning variants under the same bundle ID can produce results that reflect cached configuration rather than the current variant being tested. Initial testing for this paper produced inconsistent results that were resolved by introducing per-variant bundle identifier suffixes. Each variant installs as a distinct application, eliminating state contamination between tests. Android Network Security Configuration does not exhibit this caching behavior — testing confirmed that replacing the NSC contents and reinstalling under the same applicationId produces validation results that reflect the current configuration. The gradle product flavor approach used for the Android tests provides clean per-variant isolation but is not required to obtain accurate results.

The comparable test matrix produced expected results across sixteen test cases (eight directly-comparable variants on two platforms). Both iOS ATS and Android NSC using `pin-set`-only (note below for Android's difference) correctly enforce the configured pins: any incorrect pin causes validation failure, while correct pins permit the connection. The platforms' implementations are not the source of the security failures this paper analyzes — they enforce the configurations they're given. Here's the summary table of the tests, where Success denotes that the server could be connected, Failure denotes that the connection failed, ✓ denotes whether this was the expected and valid result, ❌ denotes where the test didn't match the expectation from the documentation:

Variant	iOS ATS Result	Android NSC Result
All pins valid	Success ✓	Success ✓

Variant	iOS ATS Result	Android NSC Result
Root valid, leaf invalid	Failure ✓	Failure ✗
Leaf valid, root invalid	Failure ✓	Failure ✗
All pins invalid	Failure ✓	Failure ✓
Root-only, valid pin	Success ✓	Success ✓
Leaf-only, valid pin	Success ✓	Success ✓
Root-only, invalid pin	Failure ✓	Failure ✓
Leaf-only, invalid pin	Failure ✓	Failure ✓

As noted in the previous paragraph, Android has two distinct elements for pinning configuration. The `pin-set` element is documented as an OR operation: "A set of public key pins. For a secure connection to be trusted, one of the public keys in the chain of trust must be in the set of pins." The `trust-anchor` element is documented as "Set of trust anchors for secure connections," which reads as *explicitly* for root-certificate-level pinning. However, without `trust-anchor`, `pin-set` becomes an AND operation — both pins must match for the server connection to be trusted. This required additional testing to verify behavior, with one additional test added for the equivalent functionality on iOS:

Variant	iOS ATS Result	Android NSC Result
`trust-anchor` **valid, leaf valid**	N/A	Success ✓
`trust-anchor` **valid, leaf invalid**	N/A	Failure ✓
`trust-anchor` **valid, one valid leaf pin, one invalid leaf pin**	Success ✓	Success ✓

This testing has demonstrated that OS-level pinning implementations enforce configurations correctly. iOS ATS and Android NSC both behave as their configurations specify, with the Android configuration-dependent composition being the one case where "as specified" requires understanding documentation gaps. The empirical work establishes the lower bound for the analysis: OS-level implementations are not the source of pinning failures. Whether third-party pinning libraries demonstrate equivalent enforcement precision is a separate

question, which the next section addresses through code review rather than empirical testing.

Validation of selected Third Party Libraries

Mobile pinning libraries often inherit engineering practices from web development contexts where the cost-to-fix assumption supports rapid iteration. The deployment economics of mobile applications differ substantially: library bugs ship to applications, get incorporated into application releases, deploy to user devices through platform update mechanisms, and require coordinated update cycles to correct. The web development cost-to-fix assumption — that issues can be addressed by deploying server-side changes — doesn't apply to library code embedded in client applications. Libraries that don't recognize this economic difference can ship code that introduces failure modes the development culture wouldn't have caught. The code reviews that follow document specific failure modes in widely-used pinning libraries, each consistent with the predictable consequences of the cultural transplant the opening identifies.

First, this section reviews Alamofire, an extremely popular and long-lived networking library on Apple platforms. It implements certificate pinning using its `PinnedCertificatesTrustEvaluator` implementation. The pinning implementation defers cryptographic validation to Security.framework, allowing Apple's framework to perform certificate chain validation. The application-level pinning logic then converts the pinned certificate set and the validated chain certificate data to `Set` types and computes `isDisjoint` between them to determine pinning success. The set-based comparison discards the hierarchical position information that distinguishes leaf from intermediate from root certificates in the validated chain. The result is OR composition: any pinned certificate matching any chain certificate satisfies validation, regardless of position. The careful deference to Security.framework for the cryptographic operations doesn't preserve the chain structure once application-level pinning logic processes the validation result.

Next, this section reviews OkHttp, a mainstay of Android mobile development and provides high-performance networking. OkHttp's certificate pinning implementation is provided through its `CertificatePinner` class. The library's documentation explicitly describes the composition semantics at the `check`

function level: the function confirms that at least one of the certificates pinned for the hostname appears in the peer certificates from the connection. This is set membership composition by definition — any single pinned certificate matching any certificate in the validated chain satisfies validation. The code implements what the documentation describes.

OkHttp's documentation references Entrust as an example of a well-known certificate authority appropriate for inclusion in a pin set. Entrust's trust status has changed significantly in the major browser root programs: Mozilla announced distrust of Entrust certificates issued after November 30, 2024, and Chrome followed with similar distrust action. Engineers consulting current OkHttp documentation may configure pin sets including Entrust based on the documentation's guidance, without awareness that Entrust's role in the public trust ecosystem has shifted. The documentation appears not to have been updated to reflect these CA trust changes, which is worth noting as engineers evaluate the documentation's currency.

The documentation also guides engineers toward configuration patterns that compound the composition's trust surface. It suggests that a typical pin set includes many root certificates from well-known certificate authorities, naming examples like Entrust and Verisign. Engineers following this guidance configure pin sets authorizing anything any of the named CAs signs across millions of certificates the CAs have issued and will issue. The composition semantics (OR) combined with the configuration guidance (multiple well-known CAs) produces a pinning configuration that resists attackers without valid certificates from any of the configured CAs while accepting anything from attackers who do have such certificates.

Finally, this section will cover TrustKit, which is available for both iOS and Android.

TrustKit's Android implementation reduces to the function `isPinInChain` in `PinningTrustManager`. The function name describes the check accurately: it returns true if any of the configured pins is found anywhere in the validated chain. The function does not check whether the pin is at a specific chain position, does not distinguish leaf from intermediate from root, and does not enforce position-specific matching. The validation outcome is determined entirely by whether at

least one pin appears somewhere in the chain. This is OR composition with chain hierarchy structurally discarded — the function as named cannot enforce anything else.

TrustKit's iOS implementation performs the validation check through `NSSet containsObject` — testing whether the computed SubjectPublicKeyInfo hash for a chain certificate appears in the configured set of known pins. The check returns success on any match regardless of which chain position produced the matching certificate. This is set membership composition with chain hierarchy discarded, structurally identical to the Android implementation's `isPinInChain` function expressed in iOS idioms.

TrustKit's configuration requires two pins to enforce pinning, one pin to configure without enforcement. The two-pin minimum encodes the assumption that legitimate pin sets contain multiple pins — typically server and backup root for rotation protection. Engineers wanting to pin to a single specific certificate cannot do so under TrustKit's configuration constraints; the library requires a second pin that then becomes another potential acceptable match through the set membership check. The configuration requirement and the composition semantics combine to ensure that pin sets always contain multiple matches, with any one match satisfying validation.

TrustKit's iOS and Android implementations are separate code bases addressing the same problem through different platform idioms. The iOS implementation uses `NSSet containsObject`; the Android implementation uses a function explicitly named `isPinInChain`. Both implementations exhibit the same architectural pattern: any configured pin matching any chain certificate satisfies validation, with chain hierarchy structurally absent from the validation question. The cross-platform consistency reflects an organizational architectural choice rather than implementation coincidence — the question "is at least one pin in the chain" is the question the library is built to ask, expressed in whatever idioms each platform supports.

As shown in this section, third-party libraries have foundational issues with their implementation of certificate pinning. They provide developers with operational flexibility that OS-level pinning doesn't directly support — programmatically

disabling pinning when the client is behind a firewall, a TLS introspection gateway, or in a debugging environment. The flexibility is real value; the architectural failures don't eliminate the operational benefit that motivates library adoption. The question is whether the operational benefit can be provided without the architectural failures.

Better Certificate Pinning

This section presents an approach to certificate pinning that improves on current SPKI-based methods across several dimensions. It is standards-based, grounded in X.509 certificate structures and the external governance documents that constrain them. Rather than build parallel validation, it consumes the validated chains TLS already produces: X.509 defines the relationships between certificates, and the approach restores the full intermediate chain that root-only or root-and-leaf SPKI pinning discards. This also makes it secure against remote forgery — the chain must validate to a root already trusted on the device, and installing a trusted root is a deliberate act requiring explicit user approval — it cannot happen silently or automatically. It is robust against the operational events that break current pinning, including certificate rotation and intermediate certificate changes within stable CA hierarchies. It scales across the population of certificates a configuration might encounter without per-certificate maintenance. And it provides relative immunity from the chaos of root certificate changes — provided the matching guidelines correctly characterize the stable patterns the configured CAs maintain through their organizational identity continuity.

These properties produce specific operational consequences worth being explicit about. Current SPKI pinning implementations impose maintenance costs at predictable intervals: every certificate rotation within an SPKI-pinned chain requires review and coordinated app updates before the rotation deadline, with support burden across the user population during the transition. These rotations are not incidental — key rotation is proper key management, and SPKI pinning breaks on precisely the practice good security requires, converting every key rotation into a maintenance event. Every CA infrastructure change that affects the pinned roots requires similar response. The CN-based approach bypasses this conflict: because CN patterns remain stable across key rotations, proper key management proceeds without breaking the pin — eliminating the key-rotation maintenance event rather than merely deferring it. Certificate rotations within

stable CA hierarchies and CA infrastructure changes within stable organizational identities likewise leave the patterns intact, removing those maintenance events as well. The remaining maintenance is monitoring for CA naming convention changes, which happen rarely if at all because the CA incentive structures discourage them, because the CA/Browser Forum Baseline Requirements outline the permitted changes. The operational cost of pinning shifts from concentrated emergency response to routine pattern verification, which is the cost difference that makes pinning sustainable for security-sensitive applications that currently can't justify the maintenance burden of SPKI approaches.

The reference implementations realize this approach as a thin layer over each operating system's own trust evaluation. They do not rebuild chains or re-derive trust; they consume the chain that the platform has already validated and apply a name-based constraint to each certificate that chain contains. The pinned quantity is the certificate's Common Name rather than its Subject Public Key Info, and the constraint is applied across the entire chain — the leaf, every intermediate, and the root. Because a Common Name is a stable identity rather than a rotating key, the volatility that forces SPKI pinning to break at every key rotation simply does not arise: the names persist across the rotations that change the keys.

This whole-chain framing is what addresses SPKI volatility at its source, because that volatility is not confined to the leaf. A chain is pinned only as well as its least stable link and narrowest accepted audience, and under SPKI every link is volatile by construction: keys are expected to rotate, and a key rotation is precisely what invalidates an SPKI pin even when the issuing organization, its role in the hierarchy, and its published name are entirely unchanged. The whole-chain constraint also narrows what the pin admits. Pinning only the root — the broadest authority in the chain — accepts every certificate that authority issues or will issue. A determined attacker does not need to forge anything: a certificate obtained from the same CA, by any means available to a well-resourced adversary, chains to the pinned root and satisfies the pin. Pinning each link, including the leaf, constrains acceptance to the specific identities that chain together for this connection. The same framing that neutralizes the volatility problem therefore also closes the over-admission that root-only pinning creates: the leaf's name distinguishes this connection from everything else the same CA signs. Pinning the name rather than the key inverts this relationship. A certificate's Common Name reflects an

organizational identity and a hierarchical role that the CA maintains deliberately and documents publicly; the key beneath that name is replaced as a matter of routine hygiene. By constraining the durable attribute and disregarding the volatile one, a CN pin remains valid across exactly the events that SPKI pinning cannot survive.

The security of a CN pin does not rest on any individual Common Name being unique or unguessable, and many are neither: an intermediate may be labeled with little more than a short mnemonic and a digit, and the root and leaf names are public by definition. What the configuration pins is not a name in isolation but the ordered set of names along the chain — the issuer-to-subject relationships that bind a leaf to its intermediates and those intermediates to a root. That relational structure is exactly what X.509 defines and what the platform's chain validation has already confirmed, which is why the approach can rest on it rather than reconstruct it. A name that would be unremarkable on its own becomes meaningful as one position in a chain of relationships the standard already formalizes; pinning the chain pins the hierarchy, not a coincidence of strings.

The implementation expresses each per-certificate constraint as one of four matching conditionals against the Common Name: `exact`, `prefix`, `prefixWithNumber`, and `suffix`. These are not interchangeable. They form a spectrum of specificity, and selecting the correct conditional for each link is where the maintenance characteristics of a configuration are decided. An `exact` match accepts a single name and nothing else — maximum specificity, and maximum maintenance, since any change to that name demands a configuration update. A `prefix` or `suffix` match accepts any name that begins or ends with a fixed fragment, trading some specificity for tolerance of the portion of the name the operator expects to vary. The `prefixWithNumber` conditional is purpose-built for the dominant form of controlled variation in CA naming: a stable mnemonic followed by an incrementing generation token. It matches a fixed prefix followed by one or more digits and nothing else, so it admits the next generation of a name while still rejecting an unrelated name that merely happens to share the prefix.

Generational naming is the case that most clearly separates the two approaches. CAs routinely issue successive generations of their issuing and root certificates and label them with a constant name and an incrementing generation marker — the

familiar "G1", "G2", "G3" sequence, or a trailing integer appended to an otherwise fixed string. Each generation is a new certificate with a new key, and each therefore invalidates any SPKI pin that named the previous generation. A CN pin written with `prefixWithNumber` against the stable portion of the name treats the generation marker as the variable it is: the same configuration that validated "G1" validates "G2" and "G3" without modification, because the pattern was pinned rather than the instance. The generational rollover that forces a coordinated SPKI update becomes, for a correctly characterized CN pin, an event the configuration already anticipated.

The same reasoning extends to root certificates, which are the most disruptive case for SPKI pinning precisely because root changes are rare, long-lived, and outside the operator's control. Crucially, root certificate names are not opaque. CAs document their roots in their Certificate Policy and Certification Practice Statements and in their published trust-store repositories, and the historical record of a CA's root names exposes the convention by which those names evolve. An operator can study that record, identify the stable mnemonic and the generational token a given CA uses, and encode a conditional that characterizes the pattern rather than the current instance. Where the convention is well established, this confers relative immunity to the next root: the pin is written to match the family of names the CA's identity continuity guarantees, and a new root that conforms to that family validates on arrival rather than triggering an outage.

Conventions do occasionally change outright, and the approach should be judged on how it behaves when they do. A CA may retire a naming family and introduce a new one, as Let's Encrypt has done in moving from the "ISRG Root X1" and "ISRG Root X2" roots toward roots named "Root YR" and "Root YE"[8]. Even this — the least favorable case — is bounded. Such roots are issued with very long lifespans, so the event is rare on the timescale of an application's maintenance; and the replacement names are not arbitrary, continuing to encode recoverable structure, here a trailing letter that denotes the key-pair type — RSA or ECDSA — exactly as the intermediate names do. A genuine convention change therefore remains both detectable and re-characterizable: it surfaces as a chain that no longer matches, and the new family can be read from the CA's documentation and folded into the configuration as a fresh pattern. It does not reopen the SPKI problem of per-

rotation churn; it substitutes a rare, scheduled re-characterization for a frequent, deadline-driven one.

None of this removes judgment, and it should not be presented as if it did. A conditional that is too loose accepts names the operator did not intend and erodes the security the pin exists to provide; a conditional that is too tight reintroduces the maintenance it was meant to remove. The objective is to make each conditional exactly as wide as the documented naming convention of the configured CA and no wider — matching the stable pattern the CA commits to and excluding everything else. Characterized correctly, the residual maintenance is the monitoring already described: watching for the rare change in a CA's naming convention, rather than responding to the routine rotation of keys beneath names that never changed.

Reference implementations of CNPinning are published at https://github.com/AustinSoftCom, with both Apple and Android versions available. They carry complete — 100% — test coverage of their security-relevant and security-adjacent code: the matching logic together with the configuration and storage structures that feed it.

The Apple implementation's library-wide coverage, as reported by Xcode's line-based coverage report, stands at 99.7%. The few unexercised lines fall into two categories, neither of which touches the accept-or-reject decision. The first category is the return in the parameterless convenience initializer: the unit-test bundle has no pinning configuration, so parsing throws `CNParseError.missingValue("CNPinningManager")` before the return is reached — and that same initializer delegates to the designated initializer that every other construction path exercises. The second is the OS-error handling inside the OSCalls abstraction layer, the defensive paths that guard against the operating system returning a value of an unexpected type — paths that, by their nature, the platform does not produce on demand. The decision logic that determines whether a connection is accepted or rejected is, in other words, covered in full, which is precisely the part whose correctness the security of the whole approach depends upon.

The Android implementation of CNPinning carries the same complete coverage of its security-relevant and security-adjacent code. Across 102 tests it reaches 100% line coverage and 99.01% branch coverage on the entire library, with the two

uncovered branches both falling outside the accept-or-reject decision. The first is a design-guarded path in the chain-link matcher: a non-empty guard short-circuits before an inlined standard-library predicate, so the empty-input branch the guard exists to prevent is structurally unreachable — the same construction as the Apple implementation, which the Android coverage tool simply counts at a finer, bytecode-level granularity. The remainder of the uncovered surface is Kotlin's compiler-generated boilerplate — the synthesized copy, componentN, equals, and hashCode members of the data classes — which no test exercises because exercising it would test the language rather than the implementation. As on Apple, the decision logic that determines whether a connection is accepted is covered in full, which is precisely the part whose correctness the security of the whole approach depends upon.

Looking back at the situation the development team found themselves in, they now have a much more robust, secure, and stable option. By documenting and enforcing the trust chain through commonName pinning across the entire chain, an attacker cannot take over the communication by obtaining a certificate from a different CA — the substituted chain would not match the pinned commonName pattern. An attacker cannot defeat the pin merely by creating a certificate chain with matching commonName patterns. The commonName match is evaluated only on a chain that has already passed normal validation — one that chains to a root in the device's trust store. A forged certificate with the right names but no valid chain to a trusted root never reaches the pinning check; it fails standard validation first. To produce a forged chain that both carries the pinned commonNames and validates to a trusted root, an attacker would need to compromise a trusted certificate authority or modify the device's trust store itself — a high-privilege, local operation, not something achievable over the network. Not impossible, but highly unlikely, and far beyond the reach of the network-level attacker that pinning is meant to defend against. The leaf certificate — potentially the most volatile credential — is now pinned by its commonName pattern rather than its SPKI, turning the volatility into something the configuration accommodates rather than breaks on. The next time the leaf is replaced, the pinning chain needs no attention. And if the servers move to a service that rotates certificates frequently, this approach survives that volatility with little effort.

[1] DigiNotar (2011) is the canonical example: the Dutch certificate authority was compromised by attackers who used the access to issue fraudulent certificates for major sites including Google, with subsequent evidence indicating use of the fraudulent certificates against Iranian internet users. The incident led to DigiNotar's bankruptcy and removal from all major root programs.

[2] Comodo reseller compromise (2011): Attackers compromised a Comodo reseller and used delegated issuance authority to obtain fraudulent certificates for major sites including login.yahoo.com, mail.google.com, and addons.mozilla.org. The compromise occurred not at Comodo's central infrastructure but through the reseller network that Comodo's commercial business model depended on for scale. The incident demonstrates that CA security cannot be evaluated solely in terms of the CA's own operations; the attack surface extends to every entity the CA has granted issuance capability, and the breadth of the delegation network reflects business pressures rather than security calculations.

[3] TURKTRUST (2013): the Turkish certificate authority that mistakenly issued two intermediate CA certificates instead of regular leaf certificates to customers who used the resulting authority to issue further certificates including for google.com.

[4] Symantec (2017-2018). A pattern of mis-issuance incidents over several years, including unauthorized test certificates issued for domains Symantec did not own, led to coordinated distrust decisions by Google and Mozilla. The eventual consequence was Symantec's sale of its CA business to DigiCert. The incident established the precedent that even the largest and most established CAs could face root program action over systematic compliance failures, with consequences that propagated across every certificate the CA had issued.

[5] Entrust distrust (2024-2025):Following years of compliance failures and mis-issuance incidents, Google announced in June 2024 that Chrome would distrust Entrust-issued TLS certificates with Signed Certificate Timestamps dated after October 31, 2024. Apple and Mozilla followed with similar distrust dates in November 2024. Every relying party with active SPKI pins targeting Entrust roots faced forced migration. The incident demonstrates that even established CAs with decades of operational history can have their issuance authority effectively revoked through root program action, with consequences cascading across every certificate they had issued.

[6] The deprecation of CN-based hostname matching is established in RFC 9525 (obsoleting RFC 6125), which directs clients to prefer Subject Alternative Name for hostname identification. The deprecation's scope is specifically the hostname-matching use case; CN content at issuer chain positions is governed by CA/Browser Forum Baseline Requirements and individual root program policies, which provide the external governance structure that distinguishes issuer-side CN reliability from leaf-side CN reliability.

[7] CA/Browser Forum Baseline Requirements Section 7.1.2.10.2 establishes that commonName MUST be present in CA certificates and SHOULD contain an identifier making the certificate's Name unique within the issuing certificate's hierarchy. The mandatory presence requirement combined with the uniqueness convention provides the foundation for CN-based identity verification at the issuer chain positions. CAs operating in publicly-trusted hierarchies populate CN with identifying content as a contractual obligation, not as an optional convention that might be omitted.

[8] Internet Security Research Group (ISRG), "*Combined Certificate Policy and Certification Practice Statement*", Version 6.1 https://letsencrypt.org/documents/isrg-cp-cps-v6.1/