IJC International Journal of Computer (IJC)
Peer-Reviewed International Journal
ISSN: 2307-4523
https://ijcjournal.org

# From Privacy Disclosure to Supply-Chain Assurance: A Layered Analytical Model for Open Swift Packages Targeting iOS

Tomasz Kubiak*

* *Corresponding author: tomasz.m.kubiak@119mail.com*

Accepted: 21 July 2026 | Published: 1 September 2026

**Abstract**

Context. Apple privacy manifests move part of the disclosure burden for data access and required-reason application programming interfaces (APIs) from the application level into a composable package artifact. Yet the existence of a declaration file can be interpreted more broadly than its evidentiary function warrants: as evidence of actual behavior, security, or low software supply-chain risk. Objective. This study develops an analytical framework that separates these inferential layers and specifies a reproducible design for future research on open Swift packages targeting iOS. Materials and methods. We conducted a directed critical synthesis of 28 verified open sources: six Apple documents, four open repository and Open Source Security Foundation (OpenSSF) infrastructure sources, two assurance frameworks, ten software supply-chain studies, and six studies of privacy labels. The unit of analysis was a claim linking an observable artifact to a permissible inference. Results. The synthesis yields a four-layer adoption model: file presence, product integration, syntactic validity, and conservative code-to-declaration consistency. It also establishes an evidence boundary between disclosure, supply-chain exposure, and observed outcomes. A repository-mining protocol is specified in which these constructs remain separate and may be compared only through explicitly observational associations. Contribution. The study clarifies the scope of privacy manifests as transparency artifacts and operationalizes a future open-data investigation without converting proxies into security outcomes. The principal limitation is that repository extraction has not yet been executed and the source corpus is analytical rather than systematic; no adoption prevalence, regression estimate, or causal effect is reported.

**Keywords:** iOS; open data; privacy manifest; repository mining; software supply chain; Swift Package Manager; transparency

## 1. Introduction

Apple's privacy-manifest regime changes the location at which part of an iOS privacy claim is expressed. Instead of leaving every declaration to the final application, the regime allows a third-party package or software development kit (SDK) to contribute structured information that Xcode can aggregate into the application's privacy report. Apple announced privacy manifests and SDK signatures at Apple's Worldwide Developers Conference 2023 (WWDC23) and subsequently required approved reasons for designated application programming interfaces (APIs) in new or updated App Store submissions from 1 May 2024 [1], [2]. The associated documentation specifies the required-reason mechanism, the placement of PrivacyInfo.xcprivacy, and the validation steps used when a manifest is invalid or incomplete [3]–[6]. This creates a technically inspectable artifact in open Swift repositories. It does not, by itself, create evidence about runtime data flows, the integrity of every binary dependency, or the probability that a package will be compromised.

That distinction is easy to lose because privacy disclosure and software supply-chain assurance share a vocabulary of trust. Both concern third-party code, incomplete observability, and the consequences of integrating components maintained outside the application team. Their evidentiary objects are nevertheless different. A privacy manifest is a declaration about specified categories of access and

collection. A dependency graph describes technical reachability. Repository security checks capture selected development practices. A vulnerability or a malicious package is an observed or verified security condition. Treating these objects as interchangeable produces two errors: a valid manifest can be mistaken for a security certificate, while a large dependency surface can be described as an attack even when no exploit or compromise has been observed.

The object of this study is the class of open Swift packages that target iOS and can therefore contribute code and resources to an application. Its subject is the inferential relationship among privacy-manifest adoption, code-to-declaration consistency, and observable indicators of software supply-chain exposure. The goal is not to estimate adoption prevalence. No repository extraction or quantitative analysis was undertaken for this study, so percentages, significance tests, and regression coefficients would be unsupported. The analysis is limited to what the verified document corpus can sustain: an analytical model and a reproducible measurement protocol.

The analysis addresses three research questions. Research Question 1 (RQ1) asks which analytical layers are required to move from the presence of PrivacyInfo.xcprivacy to a defensible claim about manifest adoption. Research Question 2 (RQ2) asks how privacy disclosure can be related to supply-chain exposure without treating either construct as a realized security outcome. Research Question 3 (RQ3) asks what open-data protocol could test these relationships reproducibly in a later repository-mining study. Together, the questions connect institutional chronology, technical taxonomy, and supply-chain theory while keeping unexecuted quantitative claims outside the argument.

The study makes a moderate but distinct contribution. First, it proposes a layered adoption model that separates file presence, product integration, syntactic validity, and conservative consistency with source references. Second, it defines an evidence boundary among disclosure artifacts, exposure and practice indicators, and observed outcomes. Third, it maps this boundary to a deterministic repository-mining design based on a pinned Swift Package Index frame, release-level analysis, Apple's documented categories, manual validation, and an explicitly non-causal comparison with disaggregated OpenSSF checks. The theoretical value lies in distinguishing transparency from assurance; the practical value lies in preventing maintainers, integrators, and researchers from overstating what a repository artifact can prove.

### 1.1. Software supply-chain exposure and assurance

Research on software supply chains supplies the first half of the analytical problem: third-party component relationships can propagate exposure, but the meaning of that exposure depends on the observable being studied. Ohm et al. catalogue malicious packages and show that open package repositories can serve as delivery channels for attacks [7]. Ladisa et al. broaden this perspective through a taxonomy of attack vectors across the open-source supply chain [8]. Both works are strongest when describing mechanisms and attacker opportunities; neither implies that ordinary dependency counts are incidents. Their relevance here is therefore conceptual: they justify measuring surfaces and practices while preserving a boundary between an indicator and an attack.

Package-ecosystem studies explain why that boundary is necessary. Zimmermann et al. demonstrate that npm forms a highly connected network in which a comparatively small set of packages can affect many downstream consumers [9]. Zahan et al. operationalize "weak links" through repository and maintainer signals, illustrating how security posture can be proxied without observing compromise directly [10]. Kikas et al. and Decan, Mens, and Grosjean show that dependency networks evolve differently across package ecosystems [11], [12]. Decan, Mens, and Claes likewise identify ecosystem-specific dependency issues [13]. The shared lesson is not that Swift must exhibit the same magnitudes as npm or other ecosystems. It is that direct dependencies, transitive reach, version constraints, and maintenance practices are separable dimensions whose distributions are likely to be ecosystem-specific.

A second group of studies focuses on temporal and vulnerability-related properties of dependencies. Cox et al. define dependency freshness as the distance between used and available versions [14]. Kula et al. show that developer responses to library updates and security advisories are shaped by more than technical availability [15]. Decan, Mens, and Constantinou analyze how known vulnerabilities propagate through an npm dependency network [16]. These studies support the use of release lag, dependency resolution, and vulnerability-oriented repository checks as analytical variables. They also warn against collapsing them into a single, unvalidated "risk score": freshness is not vulnerability, vulnerability is not exploitability, and network reach is not realized harm.

The National Institute of Standards and Technology (NIST) Secure Software Development Framework (SSDF) and Supply-chain Levels for Software Artifacts (SLSA) provide a complementary normative language. The SSDF organizes practices intended to reduce the number of vulnerabilities and mitigate their effects [17]. SLSA focuses on the integrity of software artifacts and provenance across build and distribution processes [18]. Neither framework is specific to privacy manifests or iOS, and neither offers a package-level adoption measure. Their role is to mark the difference between a disclosure artifact and broader assurance. Signatures, provenance, code review, protected branches, and controlled build processes may improve confidence in how software is produced; they do not establish that a privacy declaration is behaviorally complete.

## 1.2. Privacy disclosure and the limits of self-report

The privacy-label literature supplies the second half of the problem: structured disclosure improves legibility but remains a form of self-report. Li et al. identify the practical and conceptual difficulties developers face when creating accurate privacy nutrition labels, including ambiguous data practices and reliance on third-party libraries [19]. Zhang et al. examine the usability of iOS App Privacy Labels and show that the interface through which a disclosure is communicated affects comprehension [20]. Koch et al. investigate whether privacy labels can be kept honest by comparing declared practices with other evidence [21]. Together, these studies show that the value of a standardized declaration depends on both its usability and its relationship to independently observable behavior.

Large-scale measurement studies sharpen the same point. Lalaine operationalizes discrepancies between privacy labels and observed or inferred application behavior [22]. Scoccia et al. characterize privacy labels in the Apple ecosystem and expose the limits of app-level declarations as empirical objects [23]. Kollnig et al. situate privacy labels alongside App Tracking Transparency and report that policy and interface changes do not eliminate problems of accuracy and interpretation [24]. Their unit of analysis is the application, not the reusable package. A package manifest differs because it is composable: its declarations can be aggregated when a package becomes part of an app. That feature makes package-level inspection possible, but it also introduces a packaging question absent from many label studies. A file can exist in a repository without being included in a distributed product.

Official Apple documentation confirms that composability is intentional. Xcode aggregates manifests from included SDKs, while SDK signatures provide a mechanism for validating that code originates from the expected developer [2]. The documentation also separates the technical concerns that a manifest contains: required-reason API categories, declared reasons, collected-data categories, tracking declarations, and property-list structure [3]–[6]. This separation matters analytically. Syntactic validity is testable from a file; inclusion is testable from package-target resources; code-to-declaration consistency is partially testable through conservative static analysis. Truthfulness about runtime behavior is not established by any of these tests.

## 1.3. Research gap and analytical position

The two literatures therefore meet at a specific gap rather than at a general claim that "privacy and security are related." App-level privacy studies examine the quality and usability of disclosure but rarely treat reusable Swift packages as the unit of analysis. Package-ecosystem research examines

dependencies, attacks, maintenance, and vulnerability propagation but does not treat Apple privacy manifests as an adoption artifact. Apple's documentation defines policy and syntax but provides no population-level measurement of open-package adoption. The missing element is a model that makes the two domains comparable without merging their constructs.

The framework adopted here is an evidence-boundary approach. Every observable is paired with the strongest conclusion it can support. A manifest file can support a presence claim. Target configuration can support an integration claim. A parsed property list can support a syntactic-validity claim. Source references and declared categories can support a conservative consistency claim. Dependency and repository measures can support statements about exposure or practices. Claims about runtime behavior, legal compliance, vulnerability, or compromise require independent evidence. This framework is suitable because it turns the literature's recurring caution about self-report and proxies into explicit decision rules for analysis.

## 2. Materials and Methods

This is a document-based analytical-conceptual study. It does not report a survey, experiment, machine-learning model, content analysis of a sampled package population, or statistical association. The material is a verified corpus of 28 open sources published or captured between 2015 and 23 July 2026. It comprises six Apple policy and technical documents [1]–[6]; ten peer-reviewed studies of software supply chains and dependency ecosystems [7]–[16]; two assurance frameworks [17], [18]; six peer-reviewed studies of privacy labels [19]–[24]; two sources describing the Swift Package Index frame [25], [26]; and two OpenSSF Scorecard sources [27], [28].

Sources were included when they provided primary platform requirements, open infrastructure specifications, an assurance framework, or peer-reviewed evidence directly relevant to software supply chains or privacy disclosure. Submission guidance, discussion forums, vendor commentary without primary corroboration, search-engine journal directories, and records with unverified bibliographic identifiers were excluded. The fixed corpus limits breadth but makes the chain of evidence auditable.

**Table 1: Composition and analytical function of the verified source corpus**

| Source group | | References | Analytical function |
|---|---|---|---|
| Apple policy and technical documentation | | [1]–[6] | Institutional chronology, manifest structure, required-reason categories, validation |
| Supply-chain and dependency research | 0 | [7]–[16] | Attack mechanisms, dependency reach, freshness, vulnerabilities, repository practices |
| Assurance frameworks | | [17], [18] | Secure development, provenance, and integrity boundaries |
| Privacy-disclosure research | | [19]–[24] | Self-report accuracy, usability, and behavior–declaration consistency |
| Open registry and security-data infrastructure | | [25]–[28] | Sampling frame, repository metadata, and public security-practice checks |
| Total analytical corpus | 8 | [1]–[28] | Directed critical synthesis and conceptual modeling |

The unit of analysis was a claim linking an observable artifact or repository property to a permissible inference. Claims were extracted into six categories: institutional requirement; manifest structure; packaging integration; code-to-declaration consistency; supply-chain exposure or practice; and observed privacy or security outcome. A seventh category recorded the limitation attached to each inference. The categorization was directed rather than inductive: the categories were specified before the synthesis and applied consistently across source types.

The analytical procedure had four steps. First, the Apple documents were read as a policy-and-technical sequence. The sequence begins with the introduction of privacy manifests and SDK signatures, continues through the May 2024 approved-reason requirement, and ends with technical notes on invalid manifests and required-reason entries [1]–[6]. This step identifies what the platform requires and what Xcode is documented to aggregate. It does not establish how many packages comply.

Second, the privacy-label studies were synthesized around the relationship between declaration and independent evidence [19]–[24]. The supply-chain studies were synthesized around attack mechanisms, dependency reach, maintenance, freshness, and vulnerability propagation [7]–[16]. NIST and SLSA were used to distinguish disclosure from secure development and provenance practices [17], [18]. The comparison asked where two literatures use similar trust language for different evidentiary objects.

Third, the extracted constructs were ordered by evidentiary strength. An ordering was accepted only if each step required evidence unavailable at the preceding step. For example, file presence does not prove that Swift Package Manager includes the file in a product; inclusion does not prove that the property list is valid; validity does not prove that declared categories correspond to code; correspondence does not prove runtime completeness. This rule produced the layered model reported below.

Fourth, the model was mapped to a reproducible open-data design. The proposed frame is the Swift Package Index (SPI) PackageList pinned at commit d93e7340aafd8f6fd0cdd881cc770899c2dc8bac [25]. SPI describes an index that clones repositories, extracts package metadata, and performs real builds, while cautioning that it is not a quality filter or a complete census of every Swift package [26]. Open Source Security Foundation (OpenSSF) Scorecard provides public repository-level checks and a weekly BigQuery dataset [27], [28]. These sources make a future repository analysis feasible; in the present study they are infrastructure specifications, not analyzed observations.

Construct validity was protected by three rules. First, missing or opaque evidence is represented as unknown, not as absence. Second, supply-chain variables remain disaggregated; no bespoke composite risk score is proposed. Third, the term “association” is reserved for a future observational comparison after release-level data have been extracted. The current results are models, typologies, and protocol specifications. They are not prevalence estimates or empirical associations.

# 3. Results

## 3.1. A layered model of privacy-manifest adoption

The synthesis produced a layered model of privacy-manifest adoption. Filename presence is not treated as an endpoint; adoption is expressed as a sequence of progressively stronger claims. Figure 1 shows four layers. Layer 1 is file presence: PrivacyInfo.xcprivacy exists in the selected release tree. Layer 2 is product integration: package configuration and target paths indicate that the manifest is processed or copied into a distributed product. Layer 3 is syntactic validity: the file parses as a property list and its keys and value types are recognized under the frozen Apple documentation. Layer 4 is conservative consistency: required-reason API categories detected in inspectable source code have corresponding categories and approved-reason declarations in an integrated, valid manifest.

The hierarchy is asymmetric. Evidence at a lower layer is necessary but not sufficient for the next layer. A package cannot be classified as syntactically valid if no integrated manifest can be identified. Conversely, reaching Layer 4 does not establish complete runtime correspondence. Static source analysis may detect dead code, miss dynamically dispatched behavior, and remain blind to opaque binary targets. The model therefore ends at conservative consistency rather than “compliance,” “truthfulness,” or “safety.”

Table 2: Evidentiary layers of privacy-manifest adoption

| ayer | Construct | Observable evidence | Permissible claim | Claim not established |
|---|---|---|---|---|
| 1 | File presence | PrivacyInfo.xcprivacy in release tree | The repository contains a manifest artifact | That the distributed package includes it |
| 2 | Product integration | Target-resource mapping and path consistency | The package product contributes the manifest | That the property list is valid |
| 3 | Syntactic validity | Successful parse; recognized keys and value types | The integrated artifact is structurally interpretable | That declarations correspond to code or behavior |
| 4 | Conservative consistency | Detected API category matched to declared category/reason | Inspectable source is not obviously under-declared under the frozen lexicon | Runtime completeness, legal compliance, or security |

Figure 1 shows how a stronger manifest claim depends on evidence from each preceding layer. The dashed boundary separates manifest evidence from outcomes that require independent observation.

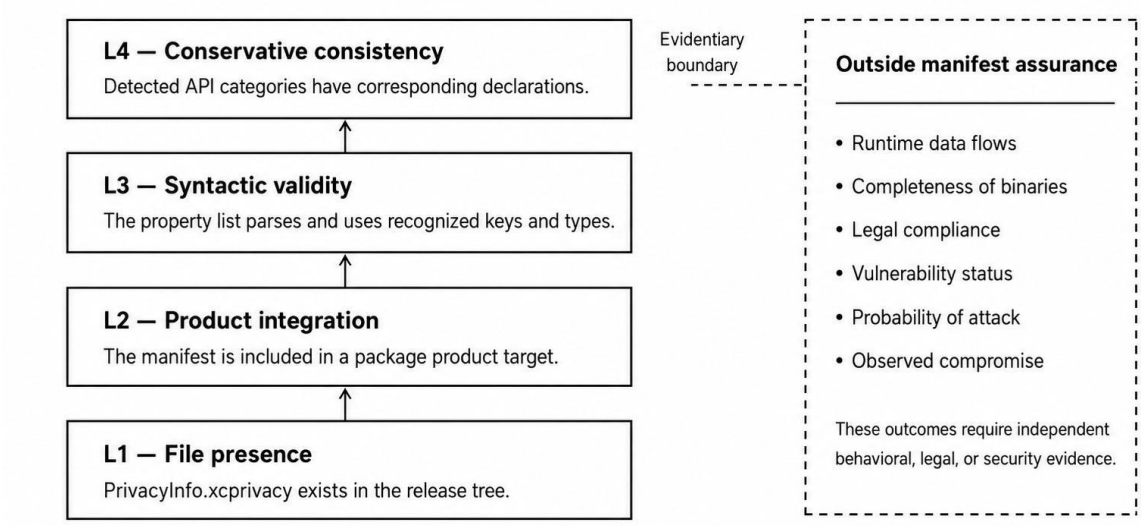


**Figure 1: Layered privacy-manifest assurance model**

*Analytical synthesis based on [2]–[6] and [19]–[24].*

The layered result therefore changes the denominator of any future adoption estimate. File-presence prevalence must use all eligible releases as its denominator, whereas integration, validity, and consistency rates must be reported as nested counts with the exclusions and unknown states retained. Reporting only the strongest state would conceal whether attrition arose from packaging, malformed syntax, or a declaration gap; reporting only file presence would overstate effective adoption.

## 3.2. The evidence boundary between disclosure and supply-chain risk

A second result is an evidence-boundary model that separates three domains often compressed into the language of "risk." Domain A contains privacy-disclosure evidence: manifest presence, integration, validity, categories, and reason codes. Domain B contains supply-chain exposure and development-practice indicators: dependency surface, mutable version requirements, remote binary targets, release freshness, signatures, provenance-related controls, and individual repository checks. Domain C contains outcomes that neither repository domain observes directly: runtime data flows, privacy harm, legal compliance, exploitable vulnerability, malicious behavior, and compromise.

Figure 2 represents the permissible relationship among the domains. Domain A and Domain B can be described separately and, after data collection, compared through an observational association. Domain C cannot be inferred directly from either. This boundary has practical consequences for

language. A repository with a remote binary target has reduced source-level observability; it has not thereby been shown malicious. A package without a manifest may be less transparent under the stated artifact model; it has not thereby been shown to collect data unlawfully. A package with a valid manifest satisfies a technical disclosure condition; it has not thereby been shown secure.

This separation also explains why a single privacy-and-supply-chain "risk score" would be a weak result. Such a score would combine constructs with different units, missingness mechanisms, directions, and normative meanings. A dependency count is a network measure; release lag is temporal; a Scorecard check is a tool-defined practice signal; a manifest state is a disclosure measure. Aggregation would create numerical precision without validated semantics. The stronger design reports each dimension and models a narrowly defined outcome—such as valid integrated manifest adoption—without renaming the predictors as incidents.

**Table 3: Construct separation and permissible inference**

| Domain | Observable evidence | Permissible inference | Inference requiring other evidence |
|---|---|---|---|
| Disclosure artifact | File, target mapping, plist keys, declared categories/reasons | Layered manifest adoption and conservative consistency | Runtime completeness; legal compliance |
| Supply-chain exposure | Direct/transitive dependencies, branch/revision requirements, remote binaries, release lag | Observable dependency and audit surface | Vulnerability; maliciousness; probability of attack |
| Repository practice | Signed releases, code review, branch protection, update tooling, provenance-oriented checks | Presence of selected development controls | Absence of flaws; effectiveness in a specific project |
| Observed outcome | Independent behavioral measurement, verified vulnerability, incident or adjudicated finding | A bounded behavior or event supported by that evidence | Generalization beyond the measured release and context |

Figure 2 states the model's central constraint. Repository evidence can support a bounded association between disclosure and practice indicators, but it cannot establish runtime, legal, or incident outcomes.

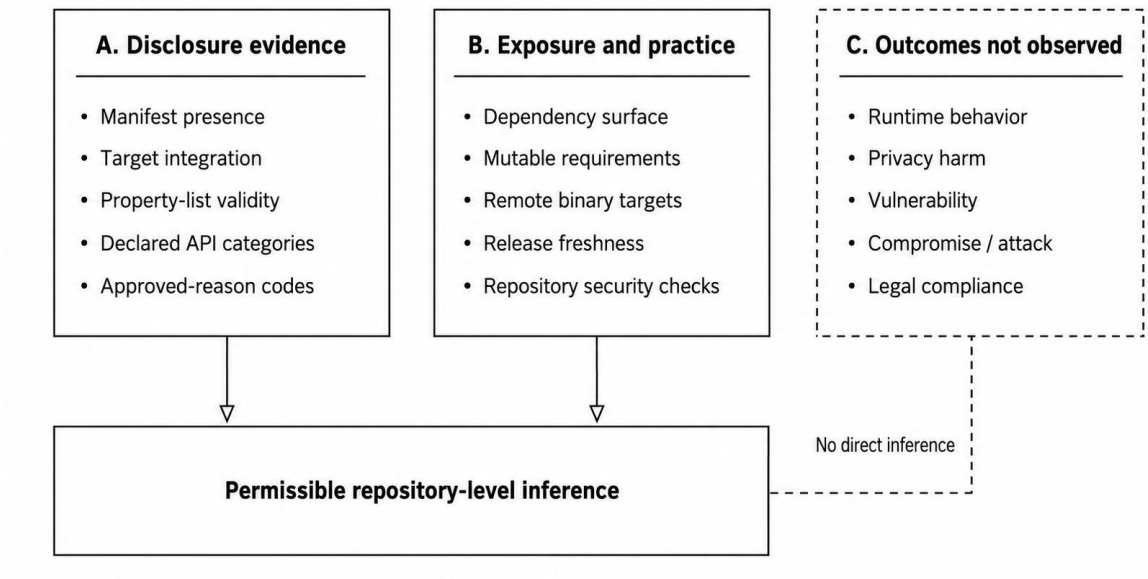


**Figure 2: Evidence boundary between disclosure, exposure, and outcomes**

*Analytical synthesis based on [7]–[24], [27], and [28].*

The boundary also yields a concrete modelling rule. If valid integrated-manifest adoption is selected as the outcome, dependency and repository-practice measures may enter as disaggregated predictors, while runtime behavior, vulnerability, and compromise must remain separate outcomes requiring

independent observation. This prevents a change in terminology from being mistaken for a change in evidence.

### 3.3. Reproducible open-data protocol

The conceptual distinction also yields a reproducible protocol for testable repository variables. The protocol is specified as future empirical work. Its target population is not "all iOS SDKs." It is open packages in a frozen Swift Package Index frame that expose a root Package.swift, have at least one stable semantic versioning (SemVer) tag, and explicitly declare iOS support. The release, rather than the current default branch, is the unit of observation because applications consume identifiable versions and because a moving branch cannot support a reproducible historical comparison.

Sampling is deterministic. Repository URLs are normalized, concatenated with the public seed PM-IOS-2026-v1, hashed with SHA-256, and sorted. Eligibility is evaluated in that order until 400 packages are included. The intended current cutoff is 22 July 2026. A paired subset uses the last stable release on or before 30 April 2024 and the last stable release on or before the current cutoff. This design can describe transitions around Apple's enforcement date, but it lacks an untreated counterfactual and therefore cannot estimate a causal policy effect.

Manifest extraction follows the four-layer model. Dependency measures, remote binaries, release lag, signed tags, code size, and owner type are collected as separate variables. OpenSSF data are joined only where a repository is covered, with individual checks retained rather than replaced by the overall score. Missing coverage remains missing. A stratified validation corpus of 80 releases—20 in each combination of automated manifest/API yes/no strata—is independently double-coded. Agreement, precision, recall, and error categories are reported where the stratified design permits them.

The planned statistical procedures remain contingent on execution. Manifest-state prevalence would be reported with Wilson confidence intervals; release transitions would be shown in a paired matrix and examined with McNemar's test; adjusted associations would use a pre-specified logistic model with valid integrated manifest as the outcome and event-per-parameter safeguards. Separation would trigger a Firth sensitivity analysis. These procedures are not results of the present article. Their inclusion makes the conceptual model falsifiable and prevents a later empirical paper from changing outcomes or predictors after observing the data.

**Table 4: Repository-mining protocol derived from the analytical model**

| Stage | Operational rule | Open basis | Primary limitation |
|---|---|---|---|
| Frame | Pinned SPI PackageList commit; normalize repository URLs | Swift Package Index / [25], [26] | Frame is reproducible but not a complete census |
| Eligibility | Public root Package.swift; stable SemVer tag; explicit iOS platform | Repository metadata | Closed SDKs and implicit iOS compatibility excluded |
| Versioning | Latest stable release by each cutoff; preserve tag and commit SHA | Git history | Dates are release proxies and can be rewritten |
| Manifest state | Presence → integration → validity → conservative consistency | Apple lexicon / [3]–[6] | Static and syntactic evidence only |
| Exposure indicators | Dependencies, mutable requirements, remote binaries, release lag, signatures | Package metadata and git | Indicators are not incidents |
| Practice indicators | Individual OpenSSF checks in covered subset | OpenSSF Scorecard / [27], [28] | Coverage is selected and may be missing not at random |
| Validation | Two coders; 80 stratified releases; adjudication and error analysis | Derived public-release corpus | Stratified validation is not automatically prevalence-weighted |
| Inference | Descriptive estimates, paired transitions, bounded associations, sensitivity analysis | Pre-specified scripts | No causal, legal, or runtime claims |

Three design choices follow from the protocol. Release-level rather than branch-level observation fixes the analyzed artifact; paired pre- and post-deadline releases distinguish within-package change from cross-sectional composition; and explicit unknown states prevent failed resolution or missing Scorecard coverage from being coded as favorable evidence. These choices improve reproducibility, although they do not create a counterfactual for causal policy evaluation.

### 3.4. Falsifiable propositions for empirical execution

Finally, four falsifiable propositions connect the model to later empirical work without presenting conjecture as evidence. Proposition P1 states that filename-only prevalence will be greater than or equal to integrated-and-valid prevalence because each successive layer imposes an additional condition. P2 states that the May 2024 deadline may be followed by upward transitions in manifest state among packages observed at both cutoffs; the proposition concerns temporal pattern, not causal effect. P3 states that declaration-gap rates will differ across required-reason API categories because APIs vary in how directly they are referenced and how easily their use is recognized in source. P4 states that valid manifest adoption may be associated with some repository practices, but no uniform direction is assumed for dependency or binary-exposure measures.

These propositions are useful precisely because they can fail. P1 would be challenged if packaging conventions make file detection less reliable than resource detection. P2 could fail if maintainers do not release after the deadline or if the current-frame survivor cohort obscures historical attrition. P3 could fail if the classifier's errors dominate category differences. P4 could fail if manifest adoption is primarily driven by App Store exposure rather than general repository maturity. The validation and sensitivity plan is therefore part of the result, not a procedural appendix: it identifies which observations would weaken the framework.

Table 5: Propositions, mechanisms, and required tests

| D | Proposition | Mechanism | Required empirical check |
|---|---|---|---|
| 1 | `Filename-only adoption ≥ integrated-and-valid adoption` | Layer nesting | Compare denominators and validated classification errors |
| 2 | Paired packages may show upward state transitions after the deadline | Institutional adoption incentive | Report transition matrix; no causal language |
| 3 | Conservative declaration gaps may vary by API category | Differences in reference visibility and declaration complexity | Category error analysis and double coding |
| 4 | Manifest adoption may correlate with selected repository practices | Shared maintenance capacity or release governance | Disaggregated observational models; no composite score |

## 4. Discussion

### 4.1. Interpretation of the principal results

The four reported outputs answer the research questions at different levels and should not be read as empirical estimates. RQ1 is answered by the four nested adoption layers: file presence, product integration, syntactic validity, and conservative consistency. RQ2 is answered by the evidence boundary, which permits disclosure and supply-chain indicators to be compared but prevents either from standing in for runtime or incident outcomes. RQ3 is answered by the release-level protocol, validation design, and pre-specified analyses. The value of these results lies in restricting what later measurements may claim, not in supplying prevalence or effect estimates before extraction.

Product integration is the pivotal layer because it is the first point at which a repository artifact becomes evidence about the package product consumed downstream. A valid PrivacyInfo.xcprivacy file outside the target-resource path is syntactically meaningful but operationally disconnected from the distributed package. This distinction explains why file searches alone can systematically overestimate adoption and why syntax checks performed before integration checks use the wrong denominator. It also gives the model diagnostic value: presence, integration, validity, and consistency failures imply different corrective actions.

The evidence-boundary result changes the formulation of a future association. A finding that packages with selected repository controls more often contain valid integrated manifests would support an association between development practice and disclosure adoption. It would not show that those packages collect less data, contain fewer vulnerabilities, or face a lower probability of compromise. Conversely, a remote binary or a wide dependency graph reduces inspectability or enlarges the audit surface, but neither observation is an incident. Keeping these meanings separate makes null, positive, and negative findings interpretable in their original units.

The protocol and propositions make the conceptual framework falsifiable. P1 tests whether the assumed nesting survives real packaging conventions and classifier error. P2 asks whether paired release states move upward after the policy deadline while explicitly allowing survival and release-cadence explanations. P3 exposes category-specific classifier limitations, and P4 tests whether manifest adoption co-varies with individual repository practices without presuming a universal direction. Failure of any proposition would refine or narrow the framework rather than being absorbed into an undifferentiated risk score.

## 4.2. Relation to previous studies

Prior privacy-disclosure research provides the closest methodological analogue, but it operates mainly at application level. Li et al. [19] document the translation difficulties developers face when assigning software behavior to standardized categories, and Zhang et al. [20] show that presentation and comprehension shape the practical value of the resulting labels. Koch et al. [21] and Xiao et al. [22] compare declarations with independent evidence and demonstrate why self-report cannot be treated as behavioral proof. Scoccia et al. [23] and Kollnig et al. [24] further show that app-level labels are useful population measures while remaining incomplete representations of behavior. The present model retains that caution but adds a package-specific condition: before validity or consistency is evaluated, the manifest must be shown to enter the product that an application actually consumes. Repository presence is therefore a weaker adoption signal for a reusable package than for a final application artifact.

Package-ecosystem studies supply measures of structure and maintenance rather than a ready-made Swift risk scale. Zimmermann et al. [9], Kikas et al. [11], and Decan et al. [12], [13] show that dependency networks have ecosystem-specific topology and evolution. Zahan et al. [10] operationalize weak links through repository and maintainer signals; Cox et al. [14] and Kula et al. [15] distinguish freshness and update behavior; and Decan et al. [16] examine the propagation of known vulnerabilities. These works justify collecting dependency depth, mutable requirements, release lag, and practice signals, but their effect sizes and thresholds cannot simply be transferred to Swift. The present synthesis therefore borrows constructs, not numerical risk meanings, and keeps each indicator disaggregated until an outcome and population have been specified.

Assurance frameworks occupy a third evidentiary level. The NIST SSDF identifies secure-development practices, while SLSA organizes artifact integrity and provenance controls [17], [18]. Apple likewise presents SDK signatures alongside privacy manifests but documents them as distinct mechanisms [2]. These sources support confidence in how software is produced and attributed; they do not validate the behavioral completeness of a privacy declaration. OpenSSF Scorecard can expose selected repository practices [27], [28], yet its checks remain tool-defined indicators with incomplete

coverage. In this study's interpretation, provenance, signatures, and repository controls can corroborate process assurance, but none can substitute for manifest integration, declaration consistency, or runtime measurement.

Across these bodies of work, the article's contribution is a translation layer rather than another universal score. It identifies which concepts can be compared across privacy engineering and software-supply-chain research, states the evidence required for each comparison, and marks the point at which additional behavioral or incident data become necessary. This boundary is the main difference from studies that treat a disclosure, a development-practice signal, and a security outcome as interchangeable manifestations of trust.

### 4.3. Practical implications

Among the four adoption layers, product integration is the most consequential methodological addition because it separates repository visibility from distributable-package evidence. Syntactic validity and conservative consistency are meaningful only after the manifest can be tied to the product that downstream applications actually consume. This ordering also clarifies the role of negative findings in future repository mining: a missing file is an absence of manifest evidence, an unintegrated file is a packaging failure under the proposed model, an invalid property list is a structural failure, and a declaration gap is a source-to-declaration inconsistency. These states should be reported separately because each points to a different remediation and supports a different inference.

For maintainers, the framework suggests a short sequence of checks. Commit the manifest at the correct package location; ensure that the product target includes it; validate the property list against the current Apple schema; and compare inspectable required-reason API references with declared categories and reasons. None of these checks should be presented as a complete runtime audit. Remote binaries require a separate trust decision because their behavior cannot be inferred from the open source tree. Signatures and provenance can strengthen the identity and integrity story but do not repair an inaccurate declaration.

For application integrators, the practical implication is to keep three review questions separate. What does the dependency declare? What is visible about how it is built and maintained? What independent evidence exists about what it does? The first question is answered partly by the manifest. The second is informed by dependency metadata, release history, signatures, provenance, and repository checks. The third requires runtime inspection, security testing, or another form of outcome evidence. A procurement or continuous integration (CI) workflow that records all three questions will be less concise than a single score, but its decisions will be auditable.

Tool builders can operationalize the first two domains while exposing uncertainty. A linter can detect PrivacyInfo.xcprivacy, parse keys, verify target-resource mapping, and flag unmatched source categories. A repository-analysis pipeline can resolve dependencies and attach individual OpenSSF checks. It should also preserve unknown values for opaque binaries, failed dependency resolution, or uncovered repositories. Converting unknown to zero would make incomplete observation look like absence of exposure and would bias both descriptive and association analyses.

## 5. Limitations of the Study

The contribution is bounded by several limitations. First, the evidence base creates a literature-selection limitation. The 28-source corpus was curated and verified for the analytical questions of this study, but it was not generated through a registered systematic-review protocol with database-wide search strings, duplicate screening, and a Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA)-style flow. Accordingly, the study supports an analytical synthesis of the verified corpus rather than a claim of exhaustive coverage, and the identified research gap should be interpreted within the stated source frame and cutoff.

Second, construct validity depends on the distinction between what is observable in a repository and what occurs at runtime. File presence, package-target integration, property-list validity, source references, dependency metadata, and repository checks are proxies with different error mechanisms. Static analysis may produce false positives through dead or unreachable code and false negatives through indirection, generated code, reflection, conditional compilation, or opaque binary targets. OpenSSF coverage is selected rather than random, and missing coverage may correlate with repository characteristics. The protocol therefore preserves unknown values, requires manual double-coding of a stratified validation set, and avoids interpreting missing evidence as evidence of absence.

Third, external and temporal validity are limited by the proposed population and by living platform rules. The framework targets open Swift packages that explicitly declare iOS support and have stable releases; proprietary SDKs, closed build systems, packages without stable tags, and behavior hidden in remote binaries are outside the frame. Apple privacy-manifest documentation and required-reason API categories can change over time, while Git tag dates are mutable proxies for release timing. A future empirical execution must therefore freeze the Apple lexicon, sampling-frame commit, package release, and analysis code at the same cutoff. Finally, the present article does not contain repository-level observations. It cannot estimate adoption prevalence, compare groups statistically, establish a policy effect, or infer effects on privacy, vulnerability, compromise, or legal compliance. Those claims require the future empirical study specified in Section 3.3.

Table 6: Limitations, affected inferences, and residual constraints

| Limitation | Affected inference | Control used in the study | Residual constraint |
|---|---|---|---|
| Curated 28-source corpus | Completeness of the literature map and research-gap claim | Fixed source categories, explicit inclusion and exclusion rules, and a stated cutoff | The synthesis is analytical rather than exhaustive or systematic |
| Repository-visible proxies | Code-to-declaration consistency and repository-practice interpretation | Layered classification, unknown values, and a stratified double-coded validation set | Runtime behavior may differ; static false positives and false negatives remain possible |
| Open Swift package frame | Generalization beyond public packages with stable releases | Pinned Swift Package Index frame and deterministic eligibility rules | Proprietary SDKs, untagged packages, closed build systems, and opaque binaries remain outside the frame |
| Living platform rules and mutable timestamps | Temporal comparability of adoption states | Freeze the Apple lexicon, frame commit, package release, and analysis code at one cutoff | Conclusions remain version-bounded; Git tag dates can be rewritten |
| Unexecuted observational design | Prevalence, association, policy effects, and security outcomes | Pre-specified denominators, transition analysis, models, and sensitivity checks | The present article supplies no empirical estimate; future associations will not by themselves be causal |

Together, these constraints define the article's interpretation envelope. The present evidence supports a bounded analytical model and a reproducible protocol for open, stable Swift packages at the stated cutoff. It does not support population prevalence, runtime completeness, legal compliance, vulnerability status, compromise probability, or a causal effect of Apple's policy. Each of those claims requires the future empirical or independent outcome evidence identified in the protocol.

## 6. Conclusion

Privacy manifests make a portion of third-party privacy disclosure visible at package level, but visibility is not equivalent to assurance. The central analytical result of this study is a boundary-preserving model: adoption progresses from file presence to product integration, syntactic validity, and conservative code-to-declaration consistency, while runtime behavior, legal compliance, vulnerability, and compromise remain outside the evidence supplied by the manifest. Supply-chain exposure and repository-practice indicators occupy a neighboring domain. They can be compared with manifest

adoption after data collection, but they cannot be renamed as outcomes or combined into an unvalidated risk score.

The study meets its objective through a falsifiable conceptual and methodological specification. It identifies the appropriate unit of future analysis, defines a deterministic open-package frame, fixes release-selection and validation rules, and states the statistical procedures that would be permissible after extraction. The contribution to empirical software engineering is an explicit account of how far a repository artifact can carry an inference. For privacy engineering, the framework clarifies where composable disclosure ends and independent behavioral evidence must begin.

The next research step is execution rather than further conceptual expansion: freeze the documented Apple lexicon, form the release-level corpus, validate the classifier, and report the layered denominators before estimating any association. A later study may then determine whether privacy-manifest adoption co-varies with selected repository practices and whether paired release transitions follow the policy deadline. Until those outputs exist, the defensible conclusion is limited but useful: privacy manifests improve the observability of declared package practices; they do not, on their own, certify software behavior or supply-chain security.

## Declarations

**Ethics approval**
Not applicable.

**Funding**
No external funding.

**Competing interests**
The author declares that there are no conflicts of interest.

**Author contributions**
The author contributed to the conception, analysis, writing, and final approval of the manuscript.

**Data availability**
No repository-level dataset was created or analyzed. All documents used in the synthesis are publicly accessible through the DOI or official URL in the reference list.

## References

[1] Apple Inc., “Privacy updates for App Store submissions,” 2024. [Online]. Available: https://developer.apple.com/news/?id=3d8a9yyh [Accessed: Jul. 23, 2026].

[2] Apple Inc., “Third-party SDK requirements.” [Online]. Available: https://developer.apple.com/support/third-party-SDK-requirements/ [Accessed: Jul. 23, 2026].

[3] Apple Inc., “Describing use of required reason API.” [Online]. Available: https://developer.apple.com/documentation/bundleresources/describing-use-of-required-reason-api [Accessed: Jul. 23, 2026].

[4] Apple Inc., “Adding a privacy manifest to your app or third-party SDK.” [Online]. Available: https://developer.apple.com/documentation/bundleresources/adding-a-privacy-manifest-to-your-app-or-third-party-sdk [Accessed: Jul. 23, 2026].

[5] Apple Inc., “TN3181: Debugging an invalid privacy manifest,” 2024. [Online]. Available: https://developer.apple.com/documentation/technotes/tn3181-debugging-invalid-privacy-manifest [Accessed: Jul. 23, 2026].

[6] Apple Inc., “TN3183: Adding required reason API entries to your privacy manifest,” 2024. [Online]. Available: https://developer.apple.com/documentation/technotes/tn3183-adding-required-reason-api-entries-to-your-privacy-manifest [Accessed: Jul. 23, 2026].

[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cham, Switzerland: Springer, 2020, pp. 23–43, doi: 10.1007/978-3-030-52683-2_2.

[8] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *2023 IEEE Symposium on Security and Privacy*, 2023, pp. 1509–1526, doi: 10.1109/SP46215.2023.10179304.

[9] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *28th USENIX Security Symposium*, 2019, pp. 995–1010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman [Accessed: Jul. 23, 2026].

[10] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What Are Weak Links in the npm Supply Chain?" in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 331–340, doi: 10.1145/3510457.3513044.

[11] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, 2017, pp. 102–112, doi: 10.1109/MSR.2017.55.

[12] A. Decan, T. Mens, and P. Grosjean, "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381–416, 2019, doi: 10.1007/s10664-017-9589-y.

[13] A. Decan, T. Mens, and M. Claes, "An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 2–12, doi: 10.1109/SANER.2017.7884604.

[14] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring Dependency Freshness in Software Systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 109–118, doi: 10.1109/ICSE.2015.140.

[15] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018, doi: 10.1007/s10664-017-9521-5.

[16] A. Decan, T. Mens, and E. Constantinou, "On the Impact of Security Vulnerabilities in the npm Package Dependency Network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 181–191, doi: 10.1145/3196398.3196401.

[17] M. Souppaya, K. Scarfone, and D. Dodson, *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*, NIST SP 800-218. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2022, doi: 10.6028/NIST.SP.800-218.

[18] SLSA, "Supply-chain Levels for Software Artifacts: Specification v1.2." [Online]. Available: https://slsa.dev/spec/v1.2/ [Accessed: Jul. 23, 2026].

[19] T. Li, K. Reiman, Y. Agarwal, L. F. Cranor, and J. I. Hong, "Understanding Challenges for Developers to Create Accurate Privacy Nutrition Labels," in *CHI Conference on Human Factors in Computing Systems*, 2022, Art. no. 588, 24 pp., doi: 10.1145/3491102.3502012.

[20] S. Zhang, Y. Feng, Y. Yao, L. F. Cranor, and N. Sadeh, "How Usable Are iOS App Privacy Labels?" *Proceedings on Privacy Enhancing Technologies*, no. 4, pp. 204–228, 2022, doi: 10.56553/popets-2022-0106.

[21] S. Koch, M. Wessels, B. Altpeter, M. Olvermann, and M. Johns, “Keeping Privacy Labels Honest,” *Proceedings on Privacy Enhancing Technologies*, no. 4, pp. 486–506, 2022, doi: 10.56553/popets-2022-0119.

[22] Y. Xiao, Z. Li, Y. Qin, X. Bai, J. Guan, X. Liao, and L. Xing, “Lalaine: Measuring and Characterizing Non-Compliance of Apple Privacy Labels at Scale,” in *32nd USENIX Security Symposium*, 2023, pp. 1091–1108. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/xiao-yue [Accessed: Jul. 23, 2026].

[23] G. L. Scoccia, M. Autili, G. Stilo, and P. Inverardi, “An Empirical Study of Privacy Labels on the Apple iOS Mobile App Store,” in *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems*, 2022, 11 pp., doi: 10.1145/3524613.3527813.

[24] K. Kollnig, A. Shuba, M. Van Kleek, R. Binns, and N. Shadbolt, “Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels,” in *2022 ACM Conference on Fairness, Accountability, and Transparency*, 2022, doi: 10.1145/3531146.3533116.

[25] Swift Package Index, “PackageList: packages.json, commit d93e7340aafd8f6fd0cdd881cc770899c2dc8bac,” 2026. [Online]. Available: https://github.com/SwiftPackageIndex/PackageList/blob/d93e7340aafd8f6fd0cdd881cc770899c2dc8bac/packages.json [Accessed: Jul. 23, 2026].

[26] Swift Package Index, “FAQ.” [Online]. Available: https://swiftpackageindex.com/faq [Accessed: Jul. 23, 2026].

[27] Open Source Security Foundation, “Scorecard: Security health metrics for open source.” [Online]. Available: https://github.com/ossf/scorecard [Accessed: Jul. 23, 2026].

[28] Open Source Security Foundation, “Scorecard checks.” [Online]. Available: https://github.com/ossf/scorecard/blob/main/docs/checks.md [Accessed: Jul. 23, 2026].