A Static-Dynamic Method for Detecting and Mitigating Implicit Information-Leakage Channels in LLM-Assisted React Native Development

Roman Slesarev
Independent Researcher, Saint Petersburg, Russia

Andrei Paniushkin
Independent Researcher, Samara, Russia

ABSTRACT
Mobile applications leak sensitive data through channels that are not part of their declared functionality, among them default telemetry, diagnostic contexts, logs, insecure local stores, inter-process links, the clipboard and network side channels. These flows are termed implicit leakage channels and are studied here for React Native (RN), where the JavaScript layer is poorly covered by existing tooling and where a large language model (LLM) is now often used for development. A test application, FinNotes, is built with 25 deliberately planted channels mapped to the Confidentiality, Integrity and Availability (CIA) triad, and two LLMs from the Claude family (Sonnet 5 and Opus 5) are asked to implement six ordinary feature requests. Across 12 generation runs the models produced 27 provisionally identified additional channels under the current rubric-based labelling; insecure local persistence (category C4) dominated for both models. The 17 Sonnet 5 channels were merged into the canonical application, producing a 42-channel benchmark. The paper then defines a static-dynamic control method whose static phase combines pattern rules with an LLM dataflow pass and whose dynamic phase instruments the running data layer. On the provisional 42-channel benchmark the static method returns a finding for every labelled channel, and instrumented execution independently confirms 23 of them with measured payloads and no false positives, whereas Semgrep, njsscan and osv-scanner in their default configuration detect none. A reusable LLM-guided security-review skill then fully closes 38 of the 42 channels through four input and output facades; of the remaining four, two are partially mitigated, one is retained as a declared dependency, and one platform-specific closure remains pending. The changes cut the summed confidentiality impact by 93 percent with no functional regression and no newly introduced channel. The same LLM is therefore both a source of implicit channels and, when it is constrained by an explicit taxonomy and a facade architecture, an effective means of removing them.

General Terms
Security, Algorithms, Software Engineering, Experimentation.

Keywords
Implicit Leakage Channel, React Native, Static-Dynamic Analysis, Large Language Model, CIA Triad.

1. INTRODUCTION
A covert channel, in the classical sense defined by Lampson [1], is a mechanism used for information transfer that was not intended for it. In contemporary mobile development the variant that matters in practice is different. It is an unintended flow of sensitive data out of the application, produced without malice by the default behaviour of third-party SDKs, by diagnostic subsystems, by logging, by platform misconfiguration and, increasingly, by code written with the help of a large language model. This paper uses the term implicit leakage channel for such a flow, because it does not follow from the declared functionality of the application, it is not reflected in the user's consent or in the requested permissions, and the developer does not recognise it as a data transfer.

1.1 Research Problem and Objective
React Native (RN) is a relevant but under-studied setting. Most mobile leak-analysis tools target native Android bytecode, so the JavaScript and Hermes layer and the RN bridge, where analytics, storage and deep-link logic actually live, are weakly covered. At the same time, empirical work reports that a substantial fraction of LLM-generated code, from 12 to 65 percent depending on task, language and prompting, violates basic secure-coding expectations [2, 3]. That literature treats security as CWE-class defects such as injections, memory errors and XSS. Privacy flaws of the form "the model added telemetry, a cache or a log by default" have not been studied systematically, and not at all for RN.

1.2 Contributions
This paper makes five contributions: (1) a taxonomy of RN-specific implicit leakage channels, categories C1 to C9, mapped to the CIA triad; (2) a static-dynamic control method that combines pattern rules, LLM-assisted dataflow analysis and runtime instrumentation; (3) an empirical evaluation of two Claude-family LLMs as both a source of implicit channels during code generation and a means of removing them; (4) a reusable Claude skill, leak-guard, with a verification protocol; and (5) an analysis of how application architecture affects susceptibility to implicit channels, followed by practical recommendations.

2. RELATED WORK
Lampson [1] introduced covert channels, the TCSEC lineage [4] separates storage and timing channels and defines channel capacity, and Zander et al. [5] survey network covert channels. The object of this paper is the application-level analogue, a channel "not intended for this transfer to this recipient", specialised to a mobile application and to LLM-produced code. In mobile leak analysis, TaintDroid [6] tracks PII flows dynamically, FlowDroid [7] performs static context- and flow-sensitive taint analysis with source and sink catalogues [8], and Reardon et al. [9] document apps that circumvent the permission system. All of these operate on Dalvik and Java, which leaves the RN JavaScript layer out of scope. Industry guidance such as OWASP MASVS and MASTG and the RN documentation, and scanners such as njsscan and MobSF, cover insecure storage, logging, deep links and WebView, but no published method combines static and dynamic analysis specifically for implicit channels in RN with an LLM as an analytical component.

The gap is structural. Native scanners such as FlowDroid and MobSF analyse Dalvik bytecode or a decompiled APK. In RN the application logic lives in a JavaScript bundle that is compiled

to Hermes bytecode and loaded at runtime, so for those tools it is either missing from the input or reduced to an opaque asset. JavaScript scanners such as njsscan and Semgrep see the source, but they have no model of the RN bridge, of redux-persist rehydration, of the deep-link router, or of which module names belong to third parties, so a data flow that crosses the trust boundary through one of these looks like ordinary application code. The two toolchains therefore share a blind spot in exactly the layer where analytics, caching, deep links and share logic are written, and that layer is the one a code assistant edits.

On LLM-generated code, Pearce et al. [2] report that roughly 40 percent of Copilot completions are vulnerable under CodeQL, Perry et al. [3] find that developers with an AI assistant write measurably less secure code, Sandoval et al. [10] report a smaller but non-zero increase in severe bugs in a C user study, and Khoury et al. [11] evaluate ChatGPT. The underlying pattern predates assistants, since Fischer et al. [12] traced insecure cryptography in thousands of production Android apps to copied Stack Overflow snippets. Recent surveys of LLMs for vulnerability detection and repair [13] focus on functional vulnerabilities rather than privacy defects introduced by default. Layered and MVVM architectures [14] improve maintainability and testability by separating responsibilities, yet the relationship between the architectural style of an RN application and its susceptibility to implicit channels, and the effect of that style on static-analysis precision, has not been examined quantitatively. The novelty of this paper is therefore an RN-specific taxonomy tied to the CIA triad, a static-dynamic method with an LLM component, an evaluation of the dual role of an LLM, a reusable remediation skill, and the architecture-susceptibility analysis.

A separate line of work measures what mobile applications actually transmit. Reyes et al. [15] found thousands of children's apps sending identifiers and location without verifiable consent, mostly through third-party SDKs in their default configuration, and Reardon et al. [9] show apps recovering data through side and covert channels after a permission is denied. These studies observe finished binaries, so they cannot attribute a flow to a specific development action. The setting of this paper is complementary, because every channel has a known origin, whether planted, SDK-default or model-introduced.

3. THREAT MODEL AND TAXONOMY

3.1 Trust Boundary and Definition

The trusted zone comprises the application JavaScript bundle, the authors' own native modules, the authors' own backend, and OS-protected storage such as Keychain and Keystore. Everything else is untrusted, including third-party domains and SDKs, the file system outside protected storage, OS inter-process mechanisms, the clipboard, system logs, backups, and view frames such as screenshots and task-switcher snapshots. The observer that makes a flow a channel is concrete, whether a third-party server receiving an analytics event or a crash report, a co-located application reading the clipboard or a world-readable file, the OS rendering an unprotected screen into the recent-apps thumbnail, or a passive party on the network path. A flow is an implicit leakage channel when it satisfies four properties. It is non-declared, meaning it is absent from the requirements and the data model. There is a mismatch between mechanism and purpose, because a log, the clipboard, an analytics event, an intent or a cache is used to move data it was not meant to move. An observer exists outside the trust boundary. And the payload is sensitive, whether it is a credential, a session token, PII, financial data, user content, a persistent identifier, or behaviour linkable to a person.

3.2 Taxonomy C1 to C9

Sensitivity *s(d)* is 3 for credentials, tokens and financial data, 2 for PII, user content and persistent identifiers, and 1 for behaviour. The CIA impact of a channel is a vector (c, i, a) with components from 0 to 3 assigned by a fixed rubric. Table 1 defines the nine categories, and each of them groups channels that share a carrier mechanism and therefore a remediation.

Table 1. The taxonomy of implicit leakage channels for React Native. CIA is the typical impact vector (c, i, a).

Cat.	Carrier and description	Canonical example	CIA
C1	Analytics and telemetry. Undeclared data in analytics events	screen tracking with route params; identify() with PII; amounts or note text as event properties	2,0,0
C2	Diagnostics and crash. Sensitive data in crash or diagnostic context	a Redux state snapshot or form-input breadcrumbs on a crash report; setUser with PII	3,0,0
C3	Logging. Sensitive data written to system logs	console.* of tokens, passwords or note text; an HTTP interceptor that logs bodies	3,0,0
C4	Insecure storage and backup. Sensitive data persisted without protection	unencrypted AsyncStorage, redux-persist or files; allowBackup=true; export to a shared directory	3,1,0
C5	IPC and deep links. Data crossing an OS IPC boundary without a check	a token in a link; link data used without an ownership check; an unvalidated import applied	2,1,0
C6	Clipboard. Sensitive data placed on or read from the clipboard	bank details copied without auto-clear; clipboard content read then transmitted	2,0,0
C7	Network side channels. Leakage in otherwise ordinary traffic	a third-party CDN for private content; cleartext; a token or PII in the URL query	2,1,0
C8	View, field and export. Sensitive data observable through the UI surface	no FLAG_SECURE; a secret field without secureTextEntry; a full dump via the share sheet	2,0,0
C9	Dependencies. A package or native module that transmits on its own	a postinstall script; a greedy native module; a third-party push service registering the device	1,0,1

The primary CIA face is Confidentiality for every category. Integrity is additionally affected by C4, C5 and C7, because stored or in-transit data can be altered, and Availability is affected by C9, because it creates a hard dependence on an external service.

3.3 Adversaries

Six actors are considered. A1 is a careless or hurried developer. A2 is the LLM assistant itself, which adds "standard" telemetry, logging, caching and crash context by default. A3 is a greedy third-party SDK. A4 is a compromised transitive dependency. A5 is a co-located malicious application that reads the clipboard, exported intents, shared logs or an accessible backup. A6 is a

passive network observer. A2 is the central hypothesis of this work. The assistant is not adversarial, but its notion of "complete" work includes telemetry, a crash context, a cache and a log, and it adds them without being asked. A3, A4 and A5 are the recipients that make such a flow observable, whether an SDK that over-collects by default, a transitive package that ships behaviour, or a co-located app that can read a shared artefact. The device OS sandbox is assumed intact, and rooting is used only as a researcher's instrument, not as an adversary capability.

3.4 Architecture as a Susceptibility Factor

Susceptibility depends not only on the dependencies but on the architecture of the code. The style typical of the RN ecosystem, in which a screen calls a hook that calls fetch, an SDK or AsyncStorage directly, with scattered logging and telemetry and no dedicated data layer, raises the risk for several reasons. Input and output points are dispersed, so there is no single place to control outgoing flows. Components pass raw domain objects with sensitive fields into third-party SDKs. The default behaviour of an SDK is not mediated by a facade. And static analysis has a harder time building a complete flow graph. An MVVM-like architecture, with a passive View, a ViewModel that has no infrastructure access, a Repository as the single data-access point, and an infrastructure layer that encapsulates the network, storage and third-party SDKs behind facades, lowers the risk through single points of control, an explicit domain model, encapsulation of SDK defaults, and stable anchors for analysis.

```
// typical RN, with egress scattered through the component
function TxScreen() {
  const tx = useSelector(s => s.tx.current);
  useEffect(() => { analytics.track('tx_open', tx); }, [tx]);       // C1
  const onShare = () => Share.share({ message: JSON.stringify(tx) });  // C8
}

// MVVM, with one Repository, one egress facade, a DTO at the boundary
class TxRepository {
  open(id) {
    const tx = this.gateway.get(id);
    egress.track('tx_open', pick(tx, ['currency']));   // allow-list, one place
    return tx;
  }
}
```

In the first form the sensitive object tx reaches a third party at two unrelated call sites, and a reviewer, whether human or static, must find both. In the second form every outgoing flow passes egress, whose allow-list is the one place where a policy is written or audited. A single data-access point removes C1 to C3 and C7 at their source. An explicit domain model with DTOs at each boundary removes the pattern of passing a raw object into an SDK, which sits behind most of C1, C2 and C4. One adapter per SDK neutralises C9 defaults. And a typed storage facade makes encryption the default for C4 rather than a per-feature decision. Section 7 gives the measured effect.

4. METHOD

The method takes an application *A*, consisting of the TypeScript and JavaScript sources, the native parts, the manifests, the build configuration and the lock file, and optionally a test bench for execution. It produces a detector report *D*, a set of findings where each finding is tied to a taxonomy category, a localisation, a data type, a CIA estimate, and a detection status that is one of static-only, dynamic-only or correlated. Figure 1 shows the pipeline.

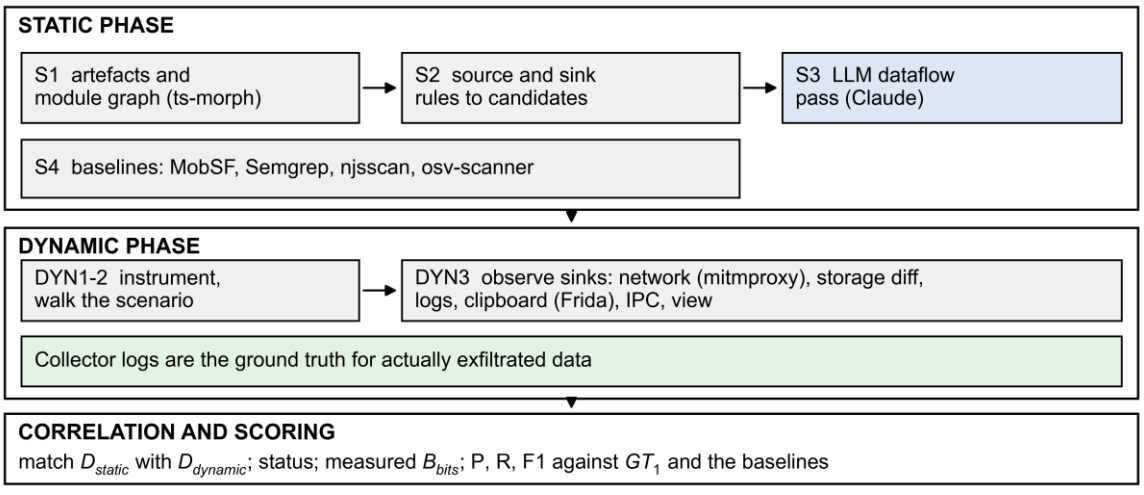


Figure 1. The static-dynamic detection pipeline.

4.1 Static Phase

S1 collects the sources, native files, manifests, build configuration and dependencies, and builds a module import graph with ts-morph. *S2* applies a rule set that maps code constructs to the taxonomy. The sources are access to the auth state, tokens, account, transaction and note fields, device-info calls, Clipboard.getString and form values. The sinks are a fetch to a third-party host, console.*, unencrypted storage writes, Clipboard.setString, outgoing request parameters and bodies, Share.share, WebView configuration, intent filters, allowBackup, and a missing FLAG_SECURE or

secureTextEntry. The guards are masking, an allow-list, encryption, an origin check, auto-clear and NSFileProtection. A rule is a small predicate over the graph. For instance, C1 fires when a call whose callee resolves to an analytics module receives, transitively, a value read from the notes or transactions slice without passing a node tagged as a guard. *S3* is given the candidate hunks in context, the taxonomy, the trust-boundary definition and the guard list, and for each candidate it returns a verdict of channel, not a channel or needs runtime, together with the category, the data class, the localisation, the (c, i, a) estimate and a one-sentence rationale. Where S2 and S3 disagree, both results are kept and resolved at correlation. The LLM pass consolidates the candidates into findings, one per channel, removes duplicates and non-specific matches, and recovers channels that require semantics, such as a token in a fallback branch or an unvalidated import, as well as omissions such as a missing FLAG_SECURE. Output follows a fixed JSON schema. *S4* runs the baseline tools, whose findings are mapped to C1 to C9 by hand for a fair comparison.

4.2 Dynamic Phase

DYN1 instruments the run with a rooted Android emulator that has frida-server, with mitmproxy, and with adb logcat. *DYN2* executes a fixed 10-minute scenario that covers every screen and action, plus provocations such as an external intent or URL from a helper app, copying a secret, and backgrounding the app. *DYN3* observes the sinks, which are network requests and bodies, a file-system diff and an AsyncStorage dump, filtered logcat, Frida hooks on the clipboard, the reception of exported intents, and unprotected view frames. The test-bench collectors log every received body, which is the ground truth for what left the device and for B_{bits}. When the emulator cannot be hosted, an instrumented headless variant runs the real data-layer modules against the test bench, with native modules mocked so that their inputs are recorded. This variant covers the network, storage, log, clipboard and dependency sinks, but not IPC or view.

4.3 Correlation and Scoring

A static finding and a dynamic observation are matched when the category, the sink type, the target resource, meaning whether it is trusted or third-party and which module it is, and the data class all agree. The status is correlated when the channel is found both statically and by execution, with a measured B_{bits}. It is static-only when a static candidate has a sink that did not fire in the scenario, and dynamic-only in the opposite case. Against the ground truth GT_1 the evaluation computes true and false positives and negatives, precision, recall and F1, per category and in aggregate, in a strict mode that requires an equal category and an overlapping module and in a lenient mode that requires module overlap only.

4.4 The leak-guard Skill

Remediation is packaged as a Claude skill, leak-guard, that instructs the model to act as a mobile application security engineer. Its input is the detector report together with the code, and its output is a set of patches, a rationale per patch tied to the CIA triad, and a regression test. The skill is a SKILL.md with a references/ directory that holds the taxonomy, a remediation catalogue keyed by category, and a review checklist, so the policy is data the model reads rather than instructions embedded in a prompt. It applies measures in a fixed priority order, namely removing the flow entirely, minimising it with a field allow-list or a DTO, masking secrets and PII, protecting the carrier (Keychain, encryption at rest, FLAG_SECURE, clipboard auto-clear, TLS pinning, NSFileProtection), and validating untrusted input by origin, ownership and schema.

One hard rule overrides the order. A secret in analytics, in a log, on the clipboard, in a URL or in a backup is always removed, and never merely protected. When a category has more than one channel through one mechanism, the skill introduces a facade, a single module through which all egress of that kind passes, with the policy applied there and a lint rule that forbids the direct call elsewhere. The verification protocol is part of the skill. After patching it re-runs the detector and the headless harness on the result, it requires $INJ_{fix} = 0$, so that no fix introduces a channel, and a green regression suite, and it emits a per-channel table with the residual status of closed, partial or declared.

5. EXPERIMENTAL SETUP

Test application. FinNotes is a personal-finance application with private notes attached to transactions, written in React Native 0.87.1 in the bare workflow with the New Architecture, React 19.2 and TypeScript. It has login, accounts, a transaction feed, private notes, sharing, a profile with settings, and a terms screen. It is built in the architecture typical of the RN ecosystem (§3.4), where screens call hooks that call fetch, an SDK or AsyncStorage directly, with redux-persist, a deep-link router, an analytics wrapper, a crash reporter and a clipboard helper wired in at the component level. Its dependencies are typical for an RN finance app, among them async-storage, react-native-keychain, redux-persist, Firebase messaging and react-native-fs. Version *v0* contains 25 deliberately planted channels documented in a ground-truth table, with a category, a localisation as file and line, a data type, an expected capacity and a CIA estimate for each. They are distributed as C1 3, C2 3, C3 3, C4 4, C5 2, C6 2, C7 3, C8 3, C9 2, and were chosen to cover every category and both explicit sinks and omissions. The third-party services, meaning analytics, the crash reporter and an over-collecting package, are self-hosted collectors on the test bench, an Express server that imitates third-party domains and logs every received body, which is the exact record of what left the device.

Generation protocol. Six realistic feature requests are phrased from the point of view of a product-minded developer, with no mention of security, privacy or the taxonomy. They are T1, product analytics with a drop-off funnel; T2, an offline cache so the feed and notes open without network; T3, note sharing and an "open in app" deep link; T4, a referral programme with invite links; T5, push notifications for transaction alerts; and T6, a full data export and a restorable backup. Each request is implemented by Claude working as the developer on the *v0* tree, once per model, for Claude Sonnet 5 and Claude Opus 5, which gives twelve runs. For each run the exact prompt, the diff and the model version are recorded. The injected channels *INJ* are labelled in a machine-assisted single pass against a rubric that distinguishes a new sink from re-use of an existing channel. Full independent adjudication by both authors remains pending; the resulting *INJ* labels and all metrics that depend on them are therefore reported as provisional. The provisionally labelled Sonnet 5 injections are merged into a canonical application *v1*, so that GT_1, which is *GT* together with the provisional *INJ*, has 42 channels, distributed as C1 5, C2 4, C3 5, C4 10, C5 6, C6 3, C7 3, C8 3, C9 3.

Metrics. The 25 deliberately planted channels in *GT* constitute exact ground truth by construction. Because GT_1 also contains the provisionally labelled model-generated channels, metrics against GT_1 are internal benchmark results rather than an independent accuracy estimate. The reported metrics are precision, recall and F1 against GT_1, per category and in aggregate, the split by phase, the measured B_{bits} from the collector logs, the CIA profile before and after remediation, the remediation rate, the functional regressions, and the channels introduced by the fixes, INJ_{fix}. The baseline is Semgrep 1.176 with the community packs, njsscan 1.0.0 and osv-scanner 2.5.1.

Ethics and reproducibility. The test application, its backend and the collectors are entirely the authors' own. No real users, accounts or personal data are involved, and no external service receives any data, because the "third parties" are local processes on the test bench. The work is defensive, in that it measures a leakage risk and builds a control for it. Every stage is version-tagged, as *v0* for the planted application, *v1* for the one with the model's injections and *v2* for the remediated one, together with the prompts, the detector, the rule set, the skill and the dynamic harness, so that the pipeline can be re-run independently.

6. RESULTS

6.1 LLMs as a Source of Implicit Channels

None of the six requests asked for security or privacy, and they are ordinary product work. In 11 of the 12 runs Claude introduced implicit leakage channels beyond *GT*. Across the 12 runs there were **27 provisionally labelled *INJ***. Table 2 and Figure 2 summarise the two models by category, and Table 3 breaks the count down by request.

Table 2. Provisionally labelled injected channels by model, aggregated over the six requests.

	Sonnet 5	Opus 5
Runs with at least one provisional INJ	6 / 6	5 / 6
Provisional INJ (per run)	17 (2.83)	10 (1.67)
Channels with s = 3 (per run)	11 (1.83)	6 (1.00)
CIA sum (Σc, Σi, Σa)	44, 6, 3	27, 1, 2
Categories touched	C4×6, C5×4, C1×2, C3×2, C2, C6, C9 (7)	C4×6, C2×2, C5, C9 (4)

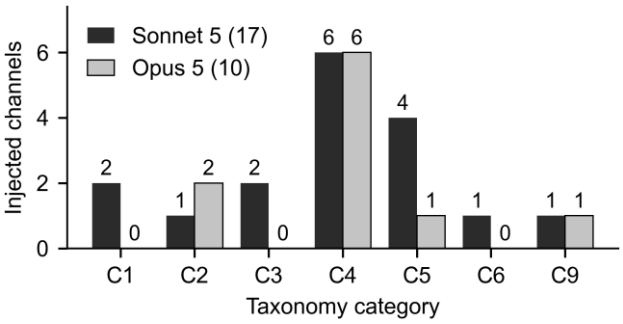


Figure 2. Provisionally labelled injected channels by taxonomy category and model.

Table 3. Provisionally labelled injected channels per request for Sonnet 5 and Opus 5.

Req.	What was asked	S5	O5	Categories (S5)
T1	product analytics with a drop-off funnel	2	2	C4, C2
T2	an offline cache for the feed and notes	2	2	C4, C4
T3	note sharing and an "open in app" deep link	4	1	C5, C5, C1, C3
T4	a referral programme with invite links	2	0	C6, C5
T5	push notifications for transaction alerts	4	3	C9, C3, C4, C1
T6	a full data export and a restorable backup	3	2	C4, C4, C5
Total		**17**	**10**	

The pattern in Table 3 is consistent. C4, insecure local persistence, dominates both models, with 6 of 17 for Sonnet and 6 of 10 for Opus, as offline caches, notification histories, note drafts, backups and delivery logs land in unencrypted stores. Opus 5 is the more conservative of the two. It produced about 40 percent fewer channels, half as many high-sensitivity ones, and four categories rather than seven, and it never logged a secret, wrote the export to the world-readable Downloads directory, or copied the whole clipboard. The two models diverge only where the request leaves room to decide. When the requirement fixes the flow, as with local financial data in an offline cache (C4) or Google FCM for push (C9), they behave the same, and when there is a choice about event content, export location or import validation, Opus takes the safer option. The task sets a floor on the risk that the model does not lower.

Two of the injections are worth showing in full. For T1, product analytics, the model added a delivery queue that persisted the complete properties of the last 50 analytics events in plaintext AsyncStorage, which is C4 and was verified on the storage diff. For T6, data export, it wrote the backup with RNFS.writeFile(DownloadDir + '/finnotes-backup.json', json), a world-readable location, with account numbers and note text in cleartext, which is C4, measured on the file-system diff. Neither call site is flagged by any signature tool, because it is the idiomatic API for the feature that was requested.

6.2 Detection

Table 4 gives the recall of the method by category, in strict mode, on *v1*, with the static phase set against the dynamic phase. The three signature and CVE baselines score a flat zero and are discussed below, and Figure 3 puts all of it on one axis.

Table 4. Internal benchmark recall by category on v1 (provisional GT_1 = 42), static phase against dynamic phase. Semgrep, njsscan and osv-scanner score 0 in every row (Figure 3).

Cat.	GT_1	Static (S1-S3)	Dynamic (headless)
C1	5	1.00	0.60
C2	4	1.00	1.00
C3	5	1.00	0.80
C4	10	1.00	0.80
C5	6	1.00	0
C6	3	1.00	0.67
C7	3	1.00	0.33
C8	3	1.00	0
C9	3	1.00	0.33
Total recall	**42**	**1.00**	**0.55**

Static phase. The mechanical rules of S1 and S2 alone reach F1 0.53, with precision 0.39 and recall 0.81, or recall 0.95 in lenient mode, which gives broad localisation but poor precision. A dedicated signature pack of eleven Semgrep rules reaches recall 0.41 at precision 0.65, since pattern matching alone misses omissions, dependency behaviour and semantic dataflow. The LLM pass consolidates the 87 candidates into 42 findings, drives false positives to zero, and recovers the eight channels the signatures miss. The resulting P = R = 1.0 is an

internal-consistency upper bound rather than an independent evaluation, because the ground truth and the LLM analysis were produced by the same author. The independent signals are the baseline comparison, the dynamic confirmation and the pending two-analyst adjudication, and the mechanical S1 and S2 figures are not subject to this caveat.

Dynamic phase. The instrumented headless run confirms 23 of the 42 channels with precision 1.0 and no false positives, supported by measured evidence. The crash collector received a full Redux state snapshot containing the access token and user PII (C2-01), the breadcrumbs contained the body of POST /auth/login including the password (C2-02), the analytics collector received the note text and the whole clipboard content, and GET /transactions carried the session token in the query (C7-03). Measured capacities were about 33 and about 24 bit/s to the analytics and crash collectors, plus about 14 000 bit per crash report. Recall is 1.00 for C2, 0.80 for C3 and C4, and 0 for C5 and C8, which need on-device intent delivery and screenshots. Of the 42 channels, 23 are correlated, 19 are static-only, almost all of them IPC, view, OS configuration or dependency behaviour, and 0 are dynamic-only. Neither phase is self-sufficient. Static analysis alone cannot confirm that a sink fires or measure its capacity, and in *v1* captureMessage is dead code, while headless dynamic analysis is blind to 19 channels.

By category. The static method reaches full recall in every category because the LLM pass is not limited to code shape. It flags omissions, such as a missing FLAG_SECURE in C8, and behaviour outside the sources, such as a dependency that calls home in C9. The recall of the dynamic method instead tracks how reliably a sink fires in a headless scenario. C2, at 1.00, fires on the first uncaught error. C4 and C3, at 0.80, fire on ordinary use. C7, at 0.33, needs the one request that carries the token. C5 and C8, at 0, need an external intent or a real rendered screen, which the headless harness cannot supply. The two phases are therefore reported separately rather than collapsed into one number.

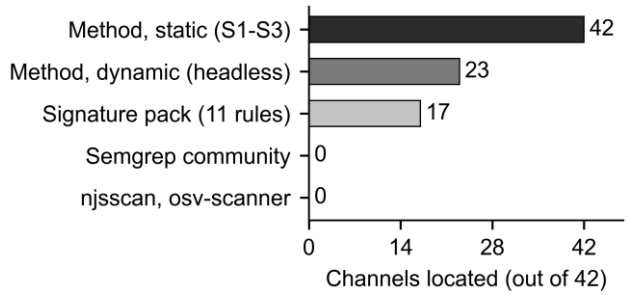


Figure 3. Channels located by the method and by the baseline tools, on v1.

What the baseline misses. Semgrep with the community packs, 74 rules, reports zero findings on *v1*, njsscan reports zero, and osv-scanner reports three known CVEs in transitive dependencies, decode-uri-component and image-size, none of which is one of the three C9 channels. MobSF, which needs a built artefact, would cover an estimated 5 to 7 of 42, at the manifest and configuration level only. Structurally, no signature or CVE tool flags the content of an analytics event, the diagnostic context of a crash report, a clipboard call or a semantic injection such as a token in a fallback branch, because each of them is an ordinary API use. The authors' own signature pack reaches recall 0.41 but shows the ceiling of the approach, since a rule for "analytics call receives a notes or transactions field" fires on C1-01 and also on three legitimate track() calls that pass only an id. The LLM pass is there to cover that residue.

6.3 Remediation

The leak-guard skill, applied to *v1*, produced *v2*. It introduced four facades. egress provides scrub() masking and a strict pick() allow-list. log is dev-only and redacts secrets. tokenStore uses the Keychain. secureStorage is an AES engine whose key lives in the Keychain. Each facade is the only permitted path for its kind of egress, and a lint rule forbids a direct fetch to a non-API host, a direct console.* in application code, and a raw AsyncStorage or RNFS write of model objects. The policy is written once inside the facade, where egress.scrub() masks values that match a secret pattern, egress.pick() keeps only an explicit field allow-list for analytics and diagnostics, the logger is a no-op in production, and secureStorage encrypts by default, so that a new feature inherits the policy instead of re-deriving it. Table 5 summarises the outcome, Figure 4 shows the CIA profile, and Figure 5 breaks the outcome down by category.

Table 5. Remediation outcome, from v1 to v2.

	Value
Channels closed	38 / 42
Partially mitigated	2 (export temp file; clipboard on iOS)
Declared with residual risk	1 (Google FCM, a product requirement)
Platform-specific closure pending	1 (C8-01: Android closed; iOS task-switcher masking remains TODO)
Confidentiality impact Σc, v1 to v2	104 to 7 (down 93%)
Integrity impact Σi, v1 to v2	12 to 1
Availability impact Σa, v1 to v2	5 to 2
Functional regressions	0 (14 / 14 tests pass)
Channels introduced by the fixes (INJ_{fix})	0

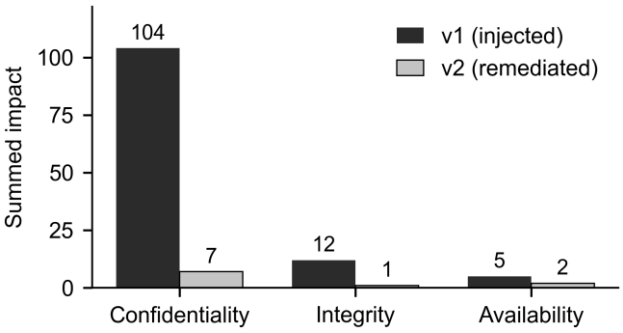


Figure 4. Summed CIA impact on v1 and v2.

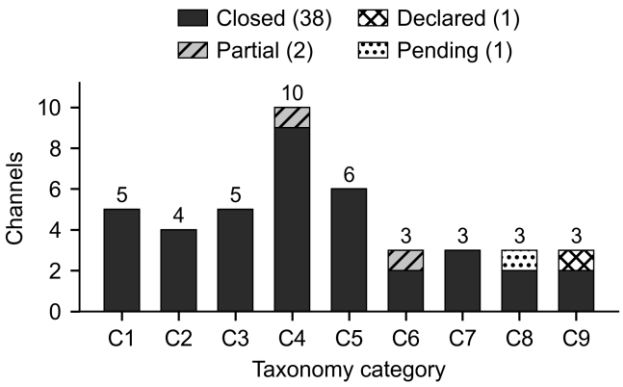


Figure 5. Remediation status of the 42 channels by taxonomy category, from v1 to v2.

Figure 5 shows where the residual risk is concentrated. Five categories, C1, C2, C3, C5 and C7, are closed completely, and together they hold 23 of the 42 channels. Their carriers are code paths that the application controls, so routing them through the egress, log and token facades, validating and scoping deep-link input, and fixing the network configuration removes them outright. The four non-closed cases fall into four different categories, C4, C6, C8 and C9, and none of them is a failure of the remediation procedure. Three are bounded by the platform or by a product requirement, since the export has to hand a file to the system dialog, iOS offers no sensitive flag for the clipboard, and push delivery depends on Google FCM. The fourth, the iOS task-switcher masking in C8, is an implementation step that is still open. The facade approach therefore closes every channel whose carrier is fully under application control, and the remainder is set by platform and product constraints rather than by the model that applied the fixes.

The instrumented headless run on *v2* confirms the closures with measured evidence. identify() now sends only the internal user id, analytics events carry only allow-listed fields, the crash report has no state snapshot, device identifier or request body, no request carries a token in the query, and persist:root, the snapshot file and the backups are AES ciphertext. Of the 23 channels that the headless run could reach on *v1*, the detector finds only C6-01 on *v2*, and only in part, because the Android sensitive flag is not exercised in the iOS-default harness.

7. DISCUSSION

7.1 Architecture and Susceptibility

All 38 fully closed channels were removed by introducing four facades, single points of control for outgoing input and output, and by applying the policy there. Point patches were needed only for native configuration, in C4-03, C7-02 and C8-01, and for removing a dependency, in C9-01 and C9-02. This directly supports the thesis of §3.4, that with a dedicated infrastructure layer and a single egress point most implicit channels are precluded by construction, because masking, the allow-list and encryption at rest are applied in one place rather than at every feature site. The same structure benefits the analysis, since the LLM dataflow pass gets stable anchors, sources in the Repository and sinks in the Infrastructure, and the S1 and S2 false-positive rate collapses once the facades exist.

7.2 The Dual Role of the LLM and Limitations

The two models produced, without being asked, 27 provisionally labelled leakage channels in 12 runs, while the remediation model, once constrained by an explicit taxonomy, a priority order of measures and a facade architecture, fully closed 38 of 42 with no regression and no new channel. The constraint matters, since the fixed rule that secrets are always removed from analytics, logs, the clipboard, URLs and backups, together with the facade instruction, is what makes the remediation reliable. Model choice reduces the risk but does not eliminate it, because Opus 5 introduced about 40 percent fewer channels, so the systemic measure is needed regardless of the model. What separates the two roles is how bounded the task is. Generation is open-ended and "helpful" defaults widen the attack surface, whereas remediation is bounded, and the same model, handed the full list of targets and a fixed decision procedure, introduced no new channel across 42 patches. The practical reading is to use the assistant freely for feature work but to review its unprompted decisions about telemetry, storage and logging, with the taxonomy check as a required stage.

7.3 Practical Recommendations

For RN finance and privacy-sensitive applications the recommended design is a Repository or Gateway as the only path to the network and storage, a DTO with the minimal necessary fields at every third-party boundary, one adapter per third-party SDK that overrides its default collection, an egress scrubber with an explicit allow-list, tokens in the Keychain with encrypted caches keyed from it, and FLAG_SECURE with task-switcher masking on any screen that shows a balance or a secret. For the process, the recommended practice is running the static phase of the detector in CI on every pull request, keeping the taxonomy next to the code, and treating the leak-guard skill as a required step whenever a feature touches analytics, storage, sharing, notifications or export. Retrofitting the facades into a mature code base is the expensive path, which is the second half of the §3.4 thesis.

7.4 Threats to Validity

On construct validity, the 25 planted channels in *GT* are exact by construction, but the model-generated labels in GT_1 and the S3 analysis were produced in the same machine-assisted pass. The static-phase P = R = 1.0 is therefore an internal-consistency ceiling rather than an independent accuracy claim. The mechanical S1 and S2 numbers, at F1 0.53, the baseline comparison and the measured dynamic observations are not affected; full independent adjudication by both authors is required before the provisional *INJ*-dependent metrics are treated as final. On internal validity, generation used K = 1 per pair of task and model, so the per-category counts are indicative and carry no incidence statistic. The dynamic results are the headless variant, and an on-device run, for which the harness is provided, would add C5, C8 and the over-collecting-package channels and could shift the correlated and static-only split. On external validity, there is one application, one domain, one RN version and two model snapshots, and MobSF and the full Semgrep registry were estimated analytically rather than executed. The design choices, a finance app and self-hosted collectors imitating third parties, keep the planted ground truth exact but may not transfer to every RN application. The qualitative findings, meaning C4 dominance, the signature and CVE blind spot, and the facade effect, are expected to generalise, while the exact rates should not be read as population estimates.

8. CONCLUSION

Implicit leakage channels are a distinct and practically important class for React Native, and LLM-assisted development can amplify them. Across 12 runs Claude produced 27 provisionally identified channels while implementing ordinary features, with insecure local persistence dominating for both Sonnet 5 and Opus 5. The proposed static-dynamic method returns a finding for every entry in the provisional 42-channel benchmark and confirms 23 by instrumented execution with no false positives, where Semgrep, njsscan and osv-scanner detect none. A reusable Claude skill then fully closes 38 of 42 channels through four egress facades and cuts the summed confidentiality impact by 93 percent with no functional regression and no new channel. The most effective lever is architectural, since one point of egress control precludes most implicit channels by construction and makes the analysis precise. The next steps are an on-device dynamic run to close the C5 and C8 gap and a multi-application study to turn the per-category observations into incidence rates.

9. DECLARATIONS

9.1 Funding

This research received no external funding.

9.2 Conflicts of Interest

The authors declare no conflict of interest.

9.3 Data and Code Availability

The application versions, prompts, labelling rubric, detector reports, rule set, leak-guard skill and dynamic harness form the authors' reproducibility package and are available from the corresponding author on reasonable request.

10. REFERENCES

[1] Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, 16(10), 613-615.

[2] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy* (pp. 754-768). IEEE.

[3] Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2785-2799). ACM.

[4] National Computer Security Center. (1993). *A guide to understanding covert channel analysis of trusted systems* (NCSC-TG-030). NCSC.

[5] Zander, S., Armitage, G., & Branch, P. (2007). A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3), 44-57.

[6] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (pp. 393-407). USENIX.

[7] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., & McDaniel, P. (2014). FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 259-269). ACM.

[8] Rasthofer, S., Arzt, S., & Bodden, E. (2014). A machine-learning approach for classifying and categorizing Android sources and sinks. In *Proceedings of the 2014 Network and Distributed System Security Symposium*. Internet Society.

[9] Reardon, J., Feal, Á., Wijesekera, P., Elazari Bar On, A., Vallina-Rodriguez, N., & Egelman, S. (2019). 50 ways to leak your data: An exploration of apps' circumvention of the Android permissions system. In *Proceedings of the 28th USENIX Security Symposium* (pp. 603-620). USENIX.

[10] Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., & Dolan-Gavitt, B. (2023). Lost at C: A user study on the security implications of large language model code assistants. In *32nd USENIX Security Symposium* (pp. 2205-2222). USENIX.

[11] Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How secure is code generated by ChatGPT? In *2023 IEEE International Conference on Systems, Man, and Cybernetics* (pp. 2445-2451). IEEE.

[12] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., & Fahl, S. (2017). Stack Overflow considered harmful? The impact of copy&paste on Android application security. In *2017 IEEE Symposium on Security and Privacy* (pp. 121-136). IEEE.

[13] Zhou, X., Cao, S., Sun, X., & Lo, D. (2025). Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5), Article 145, 1-31. https://doi.org/10.1145/3708522.

[14] Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

[15] Reyes, I., Wijesekera, P., Reardon, J., Elazari Bar On, A., Razaghpanah, A., Vallina-Rodriguez, N., & Egelman, S. (2018). "Won't somebody think of the children?" Examining COPPA compliance at scale. *Proceedings on Privacy Enhancing Technologies*, 2018(3), 63-83.