A Scalable Platform for Dynamic Privacy Analysis of iOS Applications on Physical Devices

Marcos Moran[0009−0009−4467−1782], David Rodriguez[0000−0002−0911−4608], and Jose M. Del Alamo[0000−0002−6513−0303]

Information Processing and Telecommunications Center,
Universidad Politécnica de Madrid,
ETSI Telecomunicación, Madrid, Spain
marcos.mpantoja@upm.es, david.rtorrado@upm.es, jm.delalamo@upm.es

Abstract. The runtime data practices of iOS applications remain understudied at ecosystem scale, limiting scrutiny of tracking, identifier sharing, and privacy-relevant network behavior. Such measurement is difficult because app acquisition, runtime instrumentation, and traffic attribution are constrained by Apple's ecosystem and hardware. We present an automated platform for unattended privacy analysis of iOS applications on physical devices. It combines App Store metadata retrieval and compatibility filtering, native acquisition and IPA extraction, controlled execution with pre-authorized permissions, TLS interception, and TCP-connection-level attribution of proxy-visible HTTP(S) traffic to the target executable. A queue-based manager coordinates access to jailbroken iPhones, isolates per-app failures, and scales by adding devices.
Controlled experiments with concurrent background traffic and adversarial port reuse retained all 170 target requests and excluded all 101 background requests in the evaluated TCP scenarios. A fixed batch of 50 App Store identifiers was processed with one to five devices. Expanding the pool from one to five reduced end-to-end time from 2 h 31 min to 35 min 48 s, yielding a normalized speedup of 4.14× and 82.8% parallel efficiency. Finally, an evaluation of 900 popular App Store entries produced 739 eligible applications; 621 completed normally, and app-attributed traffic was recovered for 584. The experiment finished in approximately 8 h 45 min and retained 55,501 decoded HTTP(S) flows. These results show that large-scale measurement of observable iOS data practices is feasible, enabling systematic studies of third-party communications, identifier transmission, and discrepancies between declared and observed privacy behavior.

Keywords: iOS Privacy · Dynamic Analysis · Automated Traffic Interception · SSL Pinning Bypass · App Tracking Transparency.

1 Introduction

Smartphones have become central to communication, work, entertainment, and access to digital services. Their usefulness depends largely on application ecosystems in which third-party software can access sensors, identifiers, accounts, and

network services. These capabilities also create substantial privacy risks, while app stores contain millions of applications that are continuously updated. Manual inspection is therefore insufficient for studying mobile data practices at ecosystem scale, motivating automated techniques that can observe application behavior under controlled conditions.

Although large-scale dynamic analysis has been studied extensively on Android, comparable experimentation on iOS remains considerably more difficult. Apple tightly couples software distribution, execution, and security controls to its own hardware and services. Researchers cannot rely on the combination of publicly obtainable packages, mature emulators, and broadly accessible instrumentation mechanisms available in the Android ecosystem. Consequently, the privacy-relevant data practices of iOS applications remain harder to measure systematically at ecosystem scale. This measurement gap limits empirical scrutiny of tracking, identifier sharing, and discrepancies between declared and observed data practices. Prior work has similarly noted that, despite extensive research on privacy in the Android ecosystem, comparatively less is known about iOS and the Apple App Store [11].

Three obstacles are particularly important. First, application acquisition is controlled by the App Store and depends on account state, regional availability, device compatibility, purchase history, and FairPlay-protected delivery. Tools that reproduce undocumented server-side protocols may be disrupted by changes to those protocols. Second, realistic execution requires physical devices. A scalable platform must coordinate installation, instrumentation, execution, and cleanup across several iPhones, guarantee exclusive access to each device, and prevent individual failures from blocking the remaining workload. Third, traffic captured from an iPhone is not automatically attributable to the application under study. System services and unrelated processes generate concurrent activity, while TLS interception provides application-layer visibility without necessarily identifying the process responsible for each connection. Without reliable attribution, background traffic may be incorrectly included in an application's observed behavior.

We present an automated service-based platform for unattended privacy dynamic analysis of iOS applications on physical iPhones. It integrates App Store metadata retrieval and compatibility filtering, native on-device license acquisition and IPA extraction, controlled execution with pre-authorized permissions, TLS interception, and TCP-connection-level attribution of proxy-visible HTTP(S) traffic to the target application executable. A queue-based *Manager* service coordinates another specialized services and grants workers exclusive access to devices from a shared pool. This design separates software-bound tasks from hardware-bound execution, isolates failures at the application level, and increases runtime capacity by registering additional iPhones. The platform produces application metadata, reusable IPA packages, execution records, process-associated traffic, and structured outputs for subsequent privacy studies. By making process-attributed, proxy-visible HTTP(S) observations available at scale, it supports systematic studies of third-party communications, identifier

transmission, tracking-related behavior, and discrepancies between App Privacy Labels and observed network behavior across large populations of iOS applications.

2 Related Work

Large-scale privacy analysis is most mature on Android, where application packages, emulators, and instrumentation are readily available. TraceDroid combines automated execution with network analysis [6], while complementary studies examine embedded SDKs and their configurations [15, 9]. These pipelines demonstrate the value of automation but rely on acquisition and virtualization mechanisms that do not transfer directly to iOS.

iOS research has instead used specialized static, runtime, and measurement techniques. PiOS identified potential leaks through static analysis [7]; ProtectMyPrivacy intercepted sensitive accesses on jailbroken devices [3]; and MobileAppScrutinator supported cross-platform behavioral comparison [2]. Subsequent studies measured identifier sharing and tracking [11], the effects of App Tracking Transparency [12], and discrepancies between App Privacy Labels and observed traffic [10]. Their primary objective is to characterize privacy practices rather than provide reusable infrastructure for unattended multi-device analysis.

Scalable experimentation also requires application acquisition and realistic interaction. Existing workflows often depend on previously obtained applications, manual installation, crowdsourced collection, or external download mechanisms [2, 11, 8]. AppChk crowdsources on-device network observations [8], while WhisperTest provides voice-driven UI automation without jailbreaking [13]. These systems address complementary aspects of physical-device analysis, but do not jointly integrate native acquisition, reusable IPA extraction, exclusive scheduling across a device pool, TLS interception, and traffic filtering.

Finally, device-wide captures contain traffic from system services and unrelated processes. Prior measurements reduce this interference through controlled environments and experimental isolation [2, 11, 8], but proxy-visible HTTP(S) flows do not inherently identify their originating process. Our platform combines native on-device acquisition and device-pool orchestration with correlation of decrypted proxy flows, process-filtered device captures, and host-side captures, retaining flows only when their TCP connections can be associated with the target executable.

3 Platform Design and Implementation

3.1 Design Requirements and Architecture

The platform is designed to support unattended, end-to-end dynamic analysis of iOS applications despite scarce and stateful hardware, authenticated App Store access, runtime instrumentation, and concurrent system traffic. These constraints yield five requirements: **R1**, preserve device capacity by filtering unavailable or incompatible applications before allocation; **R2**, acquire applications

through the native App Store workflow and export reusable IPA artifacts; **R3**, grant one worker exclusive device access throughout all state-changing stages; **R4**, distinguish proxy-visible TCP traffic associated with the target executable from background communications; and **R5**, isolate per-application failures and scale processing capacity by adding devices and workers.

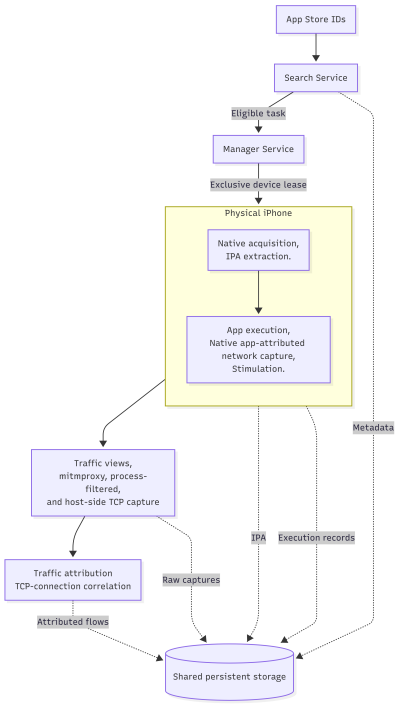

Fig. 1. Platform end-to-end analysis workflow. The *Search* service filters unavailable or incompatible applications before device allocation. The *Manager* then grants a device worker exclusive access to a physical iPhone across native acquisition, execution, traffic capture, and cleanup. The *Traffic* service correlates proxy, process-filtered, and host-side captures to retain proxy-visible HTTP(S) flows associated with the target application executable. Services communicate through asynchronous task queues and shared persistent storage.

Figure 1 presents the resulting architecture and its mapping to the analysis workflow. The platform follows a service-oriented design coordinated by a central *Manager*. Each submitted App Store identifier is represented as an independent analysis task that moves through three specialized services. The *Search* service retrieves application metadata and applies availability and compatibility checks, satisfying Requirement R1 before any hardware is reserved. Eligible tasks then enter the device-dependent part of the pipeline.

The *Manager* maintains a shared pool of registered iPhones and grants a device lease to an available worker. The lease is held across acquisition, execution, traffic capture, and cleanup, enforcing Requirement R3. Once a device has been assigned, the *Download* service triggers native license acquisition and installation on the iPhone, extracts the installed application as an IPA archive, and stores it in shared persistent storage. This stage implements Requirement R2 and separates application acquisition from subsequent runtime analysis.

The *Traffic* service prepares the device, launches the application, configures runtime instrumentation and TLS interception, and collects three complementary traffic views. Proxy-decoded HTTP(S) flows are correlated with process-filtered and host-side packet captures so that only TCP connections associated with the target executable are retained, addressing Requirement R4. The retained flows and execution metadata are then stored as structured artifacts for downstream privacy analyses.

The services communicate through asynchronous task queues and shared storage. Metadata-related workers can execute independently of the physical-device pool, while device workers process applications whenever an iPhone becomes available. Every device-dependent task includes a finalization stage that releases the device regardless of whether acquisition, instrumentation, or traffic capture succeeds. This separation of concerns provides application-level failure isolation and allows runtime throughput to scale horizontally by adding devices and workers, fulfilling Requirement R5.

3.2 Manager and Device-Pool Orchestration

The *Manager* coordinates the progression of application analyses through the platform and controls access to the physical-device pool. Each App Store identifier is represented as an independent task whose state and intermediate artifacts are maintained in shared persistent storage. Metadata retrieval and compatibility filtering are dispatched to search workers, whereas acquisition, installation, runtime execution, traffic capture, and cleanup are handled by device workers. The implementation uses Celery task queues with Redis as the message broker, allowing long-running stages to execute asynchronously while preserving their dependencies.

Separating metadata-related and device-dependent queues prevents inexpensive software-bound operations from occupying physical hardware. Search workers can retrieve and validate application metadata concurrently and independently of the number of registered iPhones. Only tasks that successfully complete these checks are submitted to the device queue, where they wait until a compatible device becomes available. This organization also prevents a slow acquisition or execution from blocking unrelated applications in the batch.

Because several stages modify device-wide state, each iPhone must be assigned exclusively to a single analysis. Before beginning a device-dependent task, a worker atomically acquires an available device from the shared pool. This operation creates an exclusive lease that is retained throughout application acquisition, IPA extraction, device preparation, runtime execution, traffic capture, and

cleanup. The worker verifies SSH connectivity before using the device and does not return it to the pool between intermediate stages. Consequently, another worker cannot simultaneously install applications, alter permission databases, or modify instrumentation on the same iPhone.

The device follows a consistent lifecycle for every application. After being acquired from the pool, it is prepared for the analysis, used by the *Download* and *Traffic* services, cleaned to reduce residual state, and finally released. Device release is performed in a finalization step regardless of the analysis outcome. This guarantees that failures during acquisition, instrumentation, app execution, or traffic processing do not leave the iPhone permanently reserved. If a device is unreachable, it is returned to the pool without completing the assignment, while the unsuccessful operation and its failure stage are recorded for subsequent inspection.

Failures are isolated at the application-task level. Operations involving external services can be retried without restarting the complete batch, and an instrumentation crash or traffic-capture error affects only the corresponding application. The remaining queued tasks continue to be assigned to available devices. This design avoids global pipeline failures and permits partial artifacts, such as an acquired IPA or traffic captured before a runtime termination, to be retained when available.

Horizontal scaling follows directly from this orchestration model. The *Manager* does not encode a fixed number of devices or a device-specific execution path. Each registered iPhone is exposed through the same pool abstraction, and each device worker follows the same task lifecycle. Adding processing capacity therefore consists of registering another prepared iPhone and starting an additional device worker; the *Search*, *Download*, and *Traffic* workflows remain unchanged. Metadata workers may likewise be replicated independently. This separation allows software-bound and hardware-bound capacity to scale independently while the queue absorbs temporary differences in application acquisition and execution times.

3.3 Search and Compatibility Filtering

The *Search* service performs all checks that do not require physical hardware before an application enters the device-dependent pipeline. Given an App Store identifier, it retrieves the metadata required by the acquisition and runtime-analysis stages, including the application name, bundle identifier, current version, minimum supported iOS version, availability status, App Store URL, and the version information used to determine its external version identifier. The resulting metadata are stored as structured artifacts and reused during acquisition, IPA extraction, installation, traffic attribution, and result organization.

The service then determines whether the application can be processed by the available device farm. Entries whose App Store records are unavailable, lack required metadata, or declare a minimum iOS version newer than that supported by the experimental devices are marked as ineligible and do not enter the device

queue. Only applications that are available, identifiable, and compatible proceed to native acquisition and runtime execution.

This early filtering stage protects the platform's scarcest resource: physical-device time. Rejecting ineligible applications before device leasing avoids unsuccessful acquisition and installation attempts, reduces unnecessary interaction with App Store services, and ensures that registered iPhones are reserved for candidates that can reach the dynamic-analysis stage.

3.4 Native Application Acquisition

The *Download* service acquires eligible applications on the physical iPhone assigned by the *Manager* and produces the IPA artifact required by the subsequent instrumentation and runtime-analysis stages. Application acquisition is a major obstacle to scalable iOS analysis because apps are distributed through Apple-controlled services rather than as publicly downloadable packages. The process depends on an authenticated Apple account, regional availability, device compatibility, purchase history, and FairPlay-protected delivery.

Some acquisition tools, such as *ipatool* [4], reproduce undocumented server-side App Store protocols to request application packages directly. Although this approach avoids interacting with a physical device, it introduces a dependency on request formats and authentication procedures that Apple may change without notice. The platform instead delegates account state, license acquisition, compatibility handling, and package delivery to the native on-device App Store workflow. This design still relies on private iOS frameworks, but reduces its exposure to changes in server-side interfaces and integrates acquisition into the same physical-device workflow used for dynamic analysis.

During device preparation, the platform deploys a custom native command-line utility, named *downloader*, on each iPhone. The utility receives the App Store identifier and external version identifier previously collected by the Search service. It loads the private `StoreKitUI` and `CoreServices` frameworks and constructs the purchase objects required to initiate the on-device acquisition workflow. The Download service invokes the utility remotely through SSH. When the acquisition process presents a user-facing purchase or download prompt, an AutoTouch script completes the required interaction.

Application installation is monitored through LaunchServices. If the download does not start, the acquisition request is retried. The service considers the operation complete only after the installed application is no longer reported as a placeholder. This prevents incomplete App Store downloads from being passed to the instrumentation pipeline.

After installation, the service queries the device application registry using the bundle identifier and device UDID to locate the corresponding `.app` bundle. It then creates a temporary `Payload` directory, places the application bundle inside it, and packages the directory as an IPA-compatible archive. The resulting artifact is transferred to the analysis host through SFTP and stored in the application's directory in shared persistent storage.

Producing a reusable IPA separates acquisition from later installation and runtime-analysis operations. The artifact can be retained together with the application metadata and experimental outputs, avoiding the need to repeat the App Store acquisition workflow whenever the same application version is reanalyzed. More broadly, native acquisition removes a major manual step from the pipeline and allows application discovery, eligibility filtering, acquisition, extraction, and execution to operate as a single unattended workflow.

3.5 Runtime Execution and Instrumentation

After acquisition, the *Traffic* service prepares the assigned iPhone and executes the application under a controlled and repeatable analysis policy. The service extracts the bundle identifier and executable name from the IPA, and configures the simulated location, runtime instrumentation, and traffic-capture components. These operations are performed while the device remains exclusively leased to the corresponding worker, preventing other analyses from modifying its state.

Runtime permissions are pre-authorized before the application is launched. The platform modifies the relevant TCC database and `locationd` authorization files to grant access to the permissions covered by the experimental profile, including App Tracking Transparency. Pre-authorizing permissions prevents system dialogs from interrupting unattended execution and places all applications in the same consent-granted configuration. This policy is intended to standardize the experimental conditions and expose behavior that may occur after permission granting; it does not represent the default state experienced by users who deny or have not yet responded to permission requests.

To increase visibility into encrypted communications, the platform combines system-level and application-level TLS interception mechanisms. SSL Kill Switch 3 [14] disables common certificate-validation paths at the operating-system level. In parallel, a custom Frida [1] script is injected into the target application and hooks certificate-pinning and trust-evaluation routines that may not be affected by the system-level bypass. The two mechanisms cover complementary validation paths. Nevertheless, applications that implement custom cryptographic protocols, native pinning checks outside the hooked routines, or hardened networking stacks may still generate traffic that cannot be decrypted.

Each application is observed during two consecutive execution windows. The first window is a 40-second first-launch period in which the application runs without synthetic input. This phase captures communications generated during startup and initial idle execution without requiring user interaction. The application is then terminated and relaunched for a 60-second input-stimulation period intended to trigger behavior beyond an untouched launch.

During the second window, an AutoTouch script generates random taps and swipes within a predefined screen region. The upper and lower screen edges are excluded to reduce unintended interaction with system gestures and device controls, and a time-based random seed determines the action sequence.

After both observation windows, the service stops the instrumentation and capture processes, records the execution outcome, and removes the application from the device. Cleanup reduces residual state before the iPhone is returned to the Manager's shared device pool.

3.6 Traffic Attribution

The *Traffic* service attributes intercepted HTTP(S) traffic to the application under analysis. Throughout this paper, a *TCP connection* denotes the transport-level association used for attribution, whereas a *flow* denotes a decoded HTTP(S) record produced by `mitmproxy` and transported over such a connection. Configuring an iPhone to use an HTTP(S) proxy provides visibility into these application-layer flows, but the resulting capture is device-wide: operating-system services, Apple platform components, and unrelated background processes may communicate through the same proxy while the target application is running. Treating all intercepted traffic as application behavior would therefore introduce false associations. To reduce this noise, the platform correlates three concurrent traffic views and retains only proxy-visible flows whose underlying TCP connections can be associated with the target application executable.

The first view is produced by `mitmproxy`. It contains decoded HTTP and HTTPS flows, including request and response metadata, timestamps, hostnames, headers, and application-layer payloads when decryption succeeds. This capture provides the information required by downstream analyses, but does not reliably identify the process that originated each connection.

The second view is a process-filtered packet trace collected on the iPhone using `ipsw` [5]. Before execution, the service extracts the executable name from the application's `CFBundleExecutable` property and configures the capture for that process. This trace identifies packets associated with the target executable, but its representation of the proxied connection cannot be matched directly to a decoded HTTP flow because the device and proxy observe different sides of the connection.

The third view is a host-side packet capture collected at the analysis host running the proxy. It provides accurate timestamps and visibility into the TCP connections accepted by `mitmproxy`. Correlating this trace with the process-filtered device capture bridges the device-side process information and the proxy-side application-layer flows.

Packet matching uses NAT-tolerant signatures rather than source and destination IP addresses alone. For each packet, the platform constructs a signature from the traffic direction, client and proxy ports, TCP sequence and acknowledgment numbers, TCP flags, and payload length. These fields remain sufficiently stable across the device and proxy observations to identify corresponding packets despite address translation and the different capture points.

A host-side TCP connection is classified as associated with the target executable when either of two conditions is met. First, the platform may observe a matching TCP `SYN`, which provides direct evidence that the target executable initiated the connection. When such a match is unavailable, the connection is

accepted only after at least two distinct packet signatures from the host-side trace match packets in the process-filtered trace. Requiring two matches reduces the risk that an isolated or coincidental packet pattern is sufficient to associate unrelated background traffic with the application.

Once process-associated TCP connections have been identified, the platform maps them to `mitmproxy` flows. A decoded flow is retained when its client port corresponds to a process-associated connection and its temporal interval overlaps the lifetime of that connection. Flows with unknown client ports, missing timestamps, or no qualifying temporal overlap are discarded rather than included as target-application traffic. The implementation records these rejection cases to support diagnosis of attribution and capture failures.

Attribution is performed at the TCP-connection level, not independently for each HTTP request. A single persistent connection may transport multiple HTTP/1.1 requests or several HTTP/2 streams. When such a connection is associated with the target executable, all proxy-decoded flows transported over it are retained. Individual HTTP/2 streams are not independently assigned to processes, since the available packet-level evidence identifies the underlying TCP connection rather than the application-layer stream that generated each request.

The mechanism also accounts for successive connections that reuse the same client port. A new `SYN` carrying a different initial sequence number is treated as the beginning of a new connection, preventing flows from separate connection lifetimes from being merged. By contrast, repeated `SYN` packets with the same initial sequence number are classified as retransmissions of the existing connection.

The current implementation deliberately adopts a conservative scope. It covers decoded HTTP(S) flows transported over TCP connections that traverse the configured proxy and can be associated with the application's main executable. UDP-based communication, including QUIC and HTTP/3, falls outside this scope. Communications that bypass the proxy, remain opaque because of certificate pinning or application-layer encryption, or use unsupported transports are likewise not retained. In addition, the process-filtered capture targets the executable declared in `CFBundleExecutable`; network activity generated by separately executing app extensions, helper processes, or system daemons may therefore be excluded. Executable association also identifies the execution context that triggered a connection, but does not by itself establish which software component constructed a request or selected its fields.

As an example of downstream processing, retained requests can be inspected for occurrences of values from a device-specific analysis profile. The current detector searches query parameters, headers, and decoded textual request bodies for literal or format-normalized matches to identifiers, device attributes, and simulated location values. These outputs are reported as observable profile-value matches rather than verified personal-data disclosures, since coincidental matches and transformed values may produce false positives and false negatives, respectively.

4 Evaluation

We evaluate the platform through three complementary experiments: a controlled validation of traffic-attribution accuracy, a fixed-candidate benchmark of horizontal scalability from one to five devices, and a 900-application study of unattended operation under a realistic workload.

4.1 Controlled Traffic-Attribution Validation

We first evaluated whether the traffic-attribution mechanism could distinguish communications associated with the target application executable from concurrent traffic generated by other processes. This validation was conducted independently of the large-corpus experiment using a controlled iOS application and a separate `curl` process. The controlled application represented the target executable configured in the process-filtered capture, whereas `curl` generated traffic that traversed the same network and proxy infrastructure but originated from an independent process.

Requests generated by both traffic sources contained unique identifiers. Server-side logs recorded the received identifiers and provided an independent ground truth for determining which requests originated from the controlled application and which were generated by the background process. The retained `mitmproxy` flows were compared with this ground truth after applying the connection-correlation procedure described in Section 3.6. This allowed target requests incorrectly discarded by the platform and background requests incorrectly retained as target-application traffic to be identified directly.

The validation covered three scenarios. In the *target-only* scenario, the controlled application generated requests without concurrent traffic from the independent process. This experiment tested whether the attribution mechanism retained traffic associated with the configured executable under otherwise controlled conditions.

In the *concurrent-background* scenario, the application and the independent `curl` process generated traffic during overlapping intervals. Both traffic sources traversed the configured proxy, causing their flows to appear together in the proxy capture. This scenario evaluated whether the process-assisted correlation could retain the application's flows while excluding communications produced by the background process.

Finally, the *adversarial port-reuse* scenario exercised successive TCP connections that reused a client port. Its purpose was to verify that connection identity was not inferred from the client port alone and that a new `SYN` with a different initial sequence number was treated as a distinct connection. This scenario specifically tested the mechanism used to prevent flows belonging to separate connection lifetimes from being incorrectly merged.

We assessed attribution at the request level using precision and recall. Precision measures the proportion of retained requests generated by the controlled application, whereas recall measures the proportion of ground-truth target requests retained by the platform. The platform retained all 170 target requests,

rejected all 101 requests generated by the independent process, and excluded three unrelated system flows observed during the experiments.

Request-level precision and recall were both 1.00. The concurrent-background result shows that sharing the proxy and observation interval did not cause background requests to be attributed to the target, while the port-reuse result confirms that flows were associated with connection lifetimes rather than client ports alone. These findings apply to the evaluated proxy-visible TCP workloads; they do not establish equivalent accuracy for UDP or QUIC, proxy-bypassing connections, separately executing components, or networking behaviors absent from the controlled scenarios.

4.2 Horizontal Scalability Benchmark

We evaluated the platform's horizontal scalability using a fixed batch of 50 App Store identifiers and device-pool sizes ranging from one to five iPhones. Every configuration received the same ordered candidate list and executed the complete end-to-end pipeline, including metadata retrieval, compatibility filtering, native on-device acquisition, IPA extraction, runtime execution, traffic capture, attribution, and cleanup. Applications were downloaded again in every configuration, and IPA artifacts produced during one run were not reused in subsequent runs.

Although every configuration received the same initial list, metadata and compatibility outcomes caused the number of applications reaching the device-dependent stage to vary slightly. We therefore base the primary scalability metrics on eligible-application throughput, defined as the number of eligible applications divided by elapsed hours. Eligibility-normalized speedup is the ratio between this throughput for a given pool size and for the single-device baseline; parallel efficiency divides that speedup by the number of devices. This normalization provides a conservative comparison by accounting for differences in the workloads that consumed physical-device time.

Device-pool utilization is the cumulative time for which devices were leased by workers divided by the total device time available during the batch. It indicates whether throughput is constrained by insufficient work assignment or by the duration and variability of device-dependent tasks.

Table 1 shows that increasing the pool from one to five devices reduced the 50-candidate batch time from 2 h 31 min 18 s to 35 min 48 s, a 76.3% reduction.

With 46–48 eligible applications across configurations, the five-device run achieved a normalized speedup of 4.14× and 82.8% efficiency. The largest absolute gain occurred from one to two devices; further devices continued to reduce completion time with diminishing marginal improvements, consistent with workload variability and finite-batch tail effects. The five-device run reached 92.5% device-pool utilization, indicating that efficiency loss was not caused by persistent device starvation.

The one- to four-device configurations used iPhone X devices, while the five-device configuration added one iPhone 8. Each pool size was evaluated once: the one- and two-device runs took place on 21 July 2026 and the remaining runs

Table 1. Horizontal-scaling results for the fixed batch of 50 App Store candidates, normalized by the applications reaching the device-dependent stage.

Devices	Time	Eligible apps	Norm. speedup	Efficiency
1	2:31:18	48	1.00	100.0%
2	1:15:02	47	1.97	98.7%
3	0:52:21	46	2.77	92.3%
4	0:42:35	47	3.48	87.0%
5	0:35:48	47	4.14	82.8%

on 17 July 2026, with seven applications changing version between those dates. The benchmark therefore supports scaling over the evaluated range but does not quantify run-to-run variability or identify when shared infrastructure would become a bottleneck.

4.3 Large-Corpus Operational Evaluation

We evaluated the platform under a realistic large-scale workload using the Spanish App Store's Top 100 rankings from nine selected categories. The rankings were collected on 13 March 2026 and produced an initial corpus of 900 unique App Store identifiers; no application was included more than once. The purpose of this experiment was to assess whether the complete platform could process a large collection of real applications without manual intervention.

Before allocating physical devices, the *Search* service retrieved the metadata required by the subsequent stages, including the bundle identifier, current version, availability status, and minimum supported iOS version. Applications whose minimum requirement exceeded the iOS version supported by the device farm were marked as incompatible. We also excluded applications that became unavailable between corpus collection and execution, or for which the required metadata could not be retrieved. This process removed 123 incompatible applications and 38 entries with unavailable or incomplete metadata, leaving 739 applications eligible for device-dependent analysis.

The experiment was conducted on 13 April 2026 using five jailbroken iPhones: four iPhone X devices and one iPhone 8. All devices ran iOS 16.7 and used *palera1n* in rootless mode. The jailbreak provided the SSH access and system-level capabilities required to modify permission databases, deploy runtime instrumentation, and obtain process-filtered packet captures. The devices were connected to a dedicated Wi-Fi network and routed HTTP(S) traffic through `mitmproxy` 8.1.1. Runtime instrumentation and automation used Frida 16.1.10, SSL Kill Switch 3 v1.4, `ipsw` 3.1.640, and AutoTouch 1.0.0.

Each iPhone was configured with a fixed experimental profile, including its network settings, identifiers, Apple account and proxy endpoint. Runtime permissions, including location and App Tracking Transparency, were pre-authorized before execution. All applications were therefore evaluated under the same consent-granted configuration.

Every eligible application followed the same execution pipeline. After native acquisition, IPA extraction and device preparation, the application was launched for a 40-second first-launch period without synthetic input. It was then terminated and relaunched for a 60-second stimulation period during which AutoTouch generated random taps and swipes within a predefined screen region.

We assessed operational behavior using application progression through the pipeline. The primary metrics were: the proportion of eligible applications successfully acquired; the proportion completing the full protocol normally; the proportions terminating because of a Frida-related failure or another traffic-service failure; and the proportion for which at least one app-attributed flow was retained. Failures were classified according to the stage and error information recorded by the platform. We additionally measured total wall-clock execution time, normal protocol completions per hour, traffic-producing applications per hour, and the total number of retained flows.

A retained flow denotes a `mitmproxy`-decoded HTTP(S) flow transported over a TCP connection associated with the target application executable under the procedure described in Section 3.6. The resulting dataset therefore excludes communications that did not traverse the configured proxy, could not be decrypted, used unsupported transports, or originated from separately executing components outside the process-filtered capture.

Figure 2 summarizes progression through the experiment. Of 739 eligible applications, 733 were acquired (99.19%), 621 completed normally (84.03%), 90 terminated because of Frida (12.18%), 22 encountered another Traffic-service failure (2.98%), and six failed during acquisition.

App-attributed traffic was recovered for 584 applications (79.03% of the 739 eligible applications): 560 normal and 24 Frida-terminated runs. Among normal completions, 90.18% produced retained traffic. Completion and traffic recovery are reported separately because a normal run may produce no observable flow, while a terminated run may yield useful partial artifacts.

Using five devices, the experiment finished in approximately 8 h 45 min, corresponding to 70.9 normal completions and 66.7 traffic-producing applications per hour, and retained 55,501 decoded HTTP(S) flows. Downstream components successfully processed these artifacts.

5 Limitations

Operational success does not imply exhaustive behavioral or network coverage. A successful run indicates only that the protocol defined in Section 4.3 completed normally; conversely, failed runs may still produce useful partial artifacts. Moreover, the absence of retained flows does not establish that an application generated no traffic. Communications may occur outside the observation windows, remain untriggered, bypass the proxy, resist decryption, or use unsupported transports. The retained dataset therefore constitutes a lower bound on proxy-visible HTTP(S) traffic. Runtime-instrumentation stability remains the primary operational bottleneck.

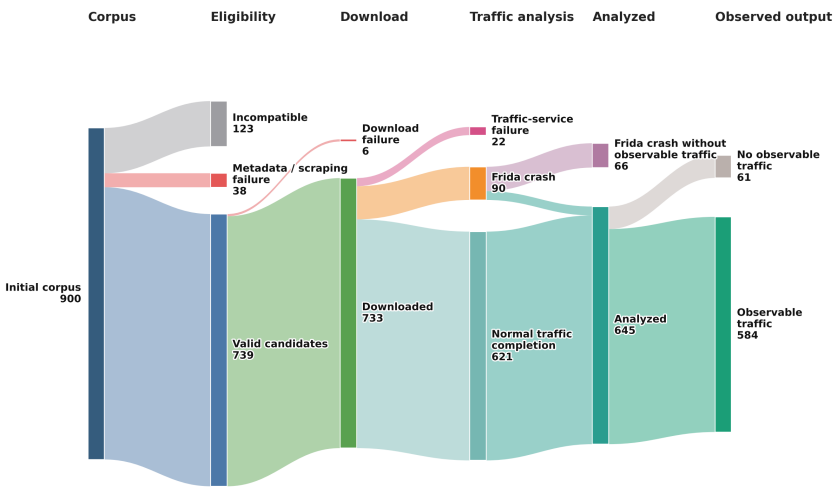

Fig. 2. Application progression through the operational evaluation. App-attributed traffic was recovered for 584 applications, including 560 normal completions and 24 Frida-terminated runs.

Attribution was validated only for proxy-visible TCP traffic associated with the main executable. The method attributes traffic at the connection level and does not independently distinguish HTTP/2 streams. It also excludes UDP/QUIC traffic, proxy-bypassing connections, extensions, helper processes, and system daemons. Furthermore, executable-level association identifies the relevant process context but not the internal component that constructed a request.

External validity is constrained by the use of jailbroken iPhones running iOS 16.7 under a consent-granted configuration. Application behavior may differ on newer iOS versions or non-jailbroken devices, while compatibility filtering may bias the corpus toward older applications. In addition, AutoTouch generates screen-agnostic random input rather than systematically exploring the UI. Consequently, it may fail to pass blocking interfaces or reach behaviorally relevant states, and its time-seeded action sequences are not directly replayable. A state-aware exploration strategy with deterministic traces could improve both coverage and reproducibility.

The scalability benchmark included a single run for each pool size, was conducted across two dates and application versions, and used heterogeneous hardware in the five-device configuration. It therefore neither quantifies run-to-run variance nor identifies the point at which shared infrastructure becomes a bottleneck. Repeated randomized experiments using homogeneous devices are necessary before extrapolating these results to larger device farms.

Reproducing the study additionally requires the platform code, candidate identifiers and metadata snapshots, device and software configurations, instrumentation and attribution scripts, failure criteria, interaction policies, timing records, and processed outputs. Redistribution of IPA files, access to Apple accounts, and proprietary App Store components may remain restricted. Nevertheless, releasing the remaining artifacts would substantially improve the repeatability of the study.

6 Conclusion

This paper presented an end-to-end platform for unattended dynamic privacy analysis of iOS applications, integrating native acquisition and IPA extraction, controlled execution, TLS interception, and queue-based orchestration of physical devices. The platform includes a validated TCP-connection-level method for automaticly identifying and retaining proxy-visible HTTP(S) flows associated with the target application executable while filtering concurrent background traffic. The evaluation further showed that the architecture can exploit additional iPhones, isolate per-application failures, and process a large corpus of real applications without manual intervention. Together, these capabilities establish the platform's operational feasibility and practical scalability as infrastructure for large-scale studies of observable iOS network behaviour. Future work can build on this infrastructure to study third-party communications and identifier transmission across large app populations, compare declared privacy practices with observed network behavior, and examine how such practices evolve across application versions and experimental consent conditions.

References

1. Frida • A world-class dynamic instrumentation toolkit, https://frida.re/
2. Achara, J.P., Roca, V., Castelluccia, C., Francillon, A.: MobileAppScrutinator: A Simple yet Efficient Dynamic Analysis Approach for Detecting Privacy Leaks across Mobile OSs (Jun 2016). https://doi.org/10.48550/arXiv.1605.08357, http://arxiv.org/abs/1605.08357, arXiv:1605.08357 [cs.CR]
3. Agarwal, Y., Hall, M.: ProtectMyPrivacy: detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In: Proceeding of the 11th annual international conference on Mobile systems, applications, and services. pp. 97–110. MobiSys '13, Association for Computing Machinery, New York, NY, USA (Jun 2013). https://doi.org/10.1145/2462456.2464460, https://dl.acm.org/doi/10.1145/2462456.2464460
4. Alfhaily, M.: majd/ipatool (Jul 2026), https://github.com/majd/ipatool, original-date: 2021-05-22T08:27:21Z
5. blacktop: blacktop/ipsw (Jul 2026), https://github.com/blacktop/ipsw, original-date: 2018-08-25T03:05:41Z
6. Cui, H., Meng, G., Zhang, Y., Wang, W., Zhu, D., Su, T., Zhang, X., Li, Y.: TraceDroid: A Robust Network Traffic Analysis Framework for Privacy Leakage in Android Apps. In: Su, C., Sakurai, K., Liu, F. (eds.) Science of

Cyber Security. pp. 541–556. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-17551-0_35
7. Egele, M., Kruegel, C., Kirda, E., Vigna, G.: Pios: Detecting privacy leaks in ios applications. In: NDSS. vol. 2011, p. 18th (2011)
8. Geier, O., Herrmann, D.: The AppChk Crowd-Sourcing Platform: Which third parties are iOS apps talking to? (Apr 2021), https://arxiv.org/abs/2104.06167v1
9. Koch, S., Karl, M., Kirchner, R., Wessels, M., Paschke, A., Johns, M.: The Impact of Default Mobile SDK Usage on Privacy and Data Protection. Proceedings on Privacy Enhancing Technologies (2025), https://petsymposium.org/popets/2025/popets-2025-0042.php
10. Koch, S., Wessels, M., Altpeter, B., Olvermann, M., Johns, M.: Keeping Privacy Labels Honest. Proceedings on Privacy Enhancing Technologies (2022), https://petsymposium.org/popets/2022/popets-2022-0119.php
11. Kollnig, K., Shuba, A., Binns, R., Kleek, M.V., Shadbolt, N.: Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. Proceedings on Privacy Enhancing Technologies (2022), https://petsymposium.org/popets/2022/popets-2022-0033.php
12. Kollnig, K., Shuba, A., Van Kleek, M., Binns, R., Shadbolt, N.: Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels. In: Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency. pp. 508–520. FAccT '22, Association for Computing Machinery, New York, NY, USA (Jun 2022). https://doi.org/10.1145/3531146.3533116, https://dl.acm.org/doi/10.1145/3531146.3533116
13. Moti, Z., Janssen-Groesbeek, T., Monteiro, S., Continella, A., Acar, G.: WhisperTest: A Voice-Control-based Library for iOS UI Automation. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 66–80. CCS '25, Association for Computing Machinery, New York, NY, USA (Nov 2025). https://doi.org/10.1145/3719027.3765183, https://dl.acm.org/doi/10.1145/3719027.3765183
14. NyaMisty: NyaMisty/ssl-kill-switch3 (Jul 2026), https://github.com/NyaMisty/ssl-kill-switch3, original-date: 2023-02-27T19:59:12Z
15. Rodriguez, D., Calandrino, J.A., Del Alamo, J.M., Sadeh, N.: Privacy Settings of Third-Party Libraries in Android Apps: A Study of Facebook SDKs. Proceedings on Privacy Enhancing Technologies (2025), https://petsymposium.org/popets/2025/popets-2025-0056.php